O'REILLY®

Broadview®
www.broadview.com.cn

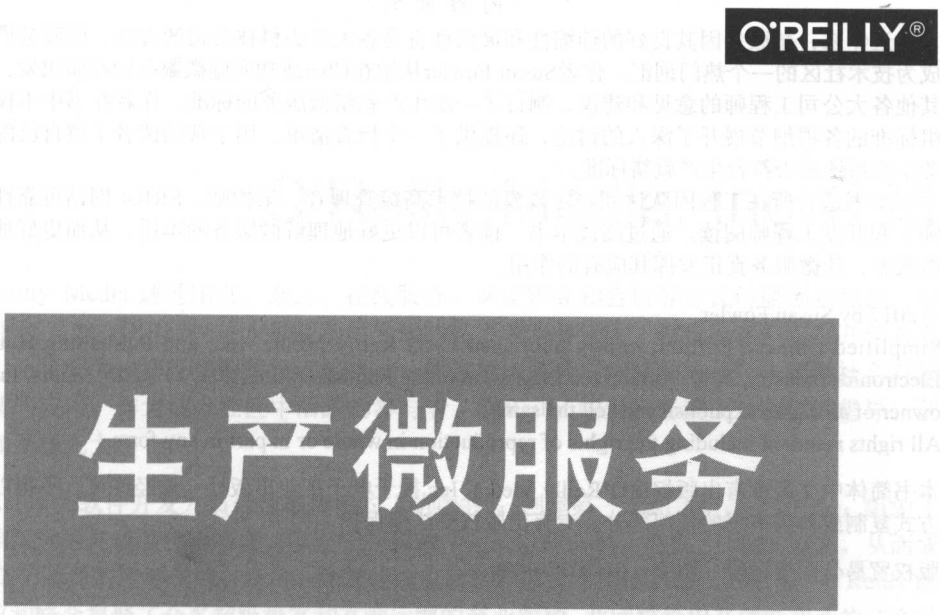
生产微服务

在工程组织范围内构建标准化的系统

Production-Ready Microservices

[美] Susan J. Fowler 著
薛命灯 译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn



Production-Ready Microservices

[美] Susan J. Fowler 著

薛命灯 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

近年来，微服务因其良好的伸缩性和灵活性备受各大巨头科技公司的青睐，微服务俨然已成为技术社区的一个热门词汇。作者Susan Fowler从她在Uber成功实施微服务的经验出发，结合其他各大公司工程师的意见和建议，制订了一组生产就绪微服务的标准。作者在书中不仅对这组标准的各项细节展开了深入的讨论，还提供了一个检查清单，用于帮助读者了解自己的微服务生态系统是否符合生产就绪标准。

本书适合所在工程团队达到一定规模的技术高级管理者、架构师、SRE（网站可靠性工程师）和开发工程师阅读。通过阅读本书，读者可以更好地理解微服务的本质，从而更好地实施微服务，让微服务真正发挥其应有的作用。

版权贸易合同登记号　图字：01-2017-0688

图书在版编目（CIP）数据

生产微服务/（美）苏珊 J. 福勒（Susan J. Fowler）著；薛命灯译. —北京：电子工业出版社，2017.9
书名原文：Production-Ready Microservices
ISBN 978-7-121-32433-8

Ⅰ. ①生… Ⅱ. ①苏… ②薛… Ⅲ. ①互联网络—网络服务器 Ⅳ. ①TP368.5

中国版本图书馆CIP数据核字（2017）第190420号

策划编辑：张春雨
责任编辑：刘　舫
封面设计：Karen Montgomery　张　健
印　　刷：三河市鑫金马印装有限公司
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787×980　1/16　　印张：9　　字数：202千字
版　　次：2017年9月第1版
印　　次：2017年9月第1次印刷
定　　价：55.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。
质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。
本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

译者序

微服务在最近几年逐渐成为一个热门的技术新名词，受到技术社区的热捧。一些巨头公司，特别是那些互联网公司，用户规模在不断增长，业务需求变得日益复杂，开发团队的规模也随之膨胀，一般的单体应用早已无法满足公司发展的需求。微服务的出现可以说是行业发展到一定阶段的必然产物。确切地说，微服务并不是一门技术，而是一种架构风格。你可以使用任何一门开发语言、任何一种框架来实现一个微服务。微服务容易开发、理解和维护，可以独立部署、独立伸缩，非常灵活。

通过将单体应用分解成微服务，解决了复杂性问题。每个微服务负责处理单一的任务，微服务之间通过定义好的接口相互通信，最后组成一个庞大的微服务生态系统。看似我们绕了一个大圈子，其实则不然。

每个微服务就是一个独立运行的应用，分别由专门的团队负责开发，开发人员可以自由选择他们熟悉的技术，也可以采用最新的技术，而且可以快速做出变更。所以对于开发人员来说，微服务给他们带来了极大的自由度，同时极大地提升了开发速度。

每个微服务可以独立开发、独立部署，而不像单体应用那样牵一发而动全身。每个微服务可以独立演化，在快速做出变更后进行部署，如果有必要，每天可以进行多次部署，因为微服务体积小，所以构建时间短，部署起来也非常方便。

每个微服务都可以独立伸缩，可以根据具体情况为每个微服务部署不同数量的实例，也可以为不同的微服务选择不同的硬件。比如，对于不是很关键的微服务可以使用便宜的硬件，对于负载不是很高的微服务就可以少部署几个实例。而对于高负载的关键微服务则多部署一些实例，并使用更好的硬件。

不过，采用微服务架构的门槛其实是很高的。Martin Fowler 认为，一个公司要采用微服务，必须满足三个基本前提条件，即快速配置能力、基本的监控能力和快速部署能力。而除此之外，要成功实施微服务，还有其他很多重要的因素需要考虑。作为 Uber 的网站可靠性工程师，Susan Fowler 在 Uber 内部致力于微服务的标准化，制订生产就绪微服务的

标准，并帮助微服务团队成功实施微服务。Susan 基于她在 Uber 成功实施微服务的经验，并结合她与其他公司工程师之间就微服务话题进行的讨论，总结出了一套生产就绪微服务的标准。本书列出的一组生产就绪微服务的检查清单可以作为成功实施微服务的参考标准。

不过话说回来，在软件技术领域并不存在什么银弹。微服务并不适合所有公司，在考虑是否采用微服务之前要先了解清楚自己的问题。先仔细想清楚，你的问题一定只能通过微服务来解决吗？如果是，那么你具备了实施微服务的条件了吗？不要只是因为那些巨头公司采用了微服务就盲目崇拜他们，如果走错了路，到最后只会给你带来惨痛的教训。

这不是一本描写具体技术实现的书，没有代码，没有具体的开发框架。但是它也不是只空讲理论，本书列出的生产就绪微服务的标准完全来自于 Uber 和其他公司的最佳实践，而且从目前来看，可以说是“前无古人，后无来者”的一次针对实施微服务的大总结。这本书值得所有的技术总监、架构师、网站可靠性工程师和开发工程师一读。先抛开脑子里的代码、开发框架，用宏观的视角审视微服务，了解微服务的本质。所谓“知己知彼，百战不殆”，只有了解了微服务的本质，才能不被其左右。当然，如果你真的需要微服务，而且具备了实施微服务的条件，那么这本书一定会给你带来不可限量的惊喜！

薛命灯

2017 年 6 月于上海

译者简介

薛命灯，毕业于厦门大学软件学院，具有十余年软件开发和架构经验。技术涉猎十分广泛，从前端到后端，从各种编程语言到分布式软件架构，从企业应用到大数据。在工作之余，爱好摄影和技术翻译，是 InfoQ 的优秀社区编辑。

目录

前言

在作为网站可靠性工程师（SRE）加入到 Uber 工作之后，我提出了生产就绪标准的倡议，并在 Uber 实施了几个月，这本书也随之诞生。Uber 庞大的单体 API 正逐渐被分解成微服务，在我加入 Uber 那会儿，已经有上千个从单体 API 分离出来的微服务，它们独立于单体系统运行。每个微服务都有专门的开发团队进行设计、开发和维护，但 85% 的微服务几乎没有 SRE。

招聘 SRE 和打造 SRE 团队不是一件容易的事情，SRE 比其他类型的工程师更难找：网站可靠性工程师是一种新型职位，SRE 必须（至少在一定程度上）是软件工程、系统工程和分布式系统架构方面的专家。在短时间内为所有的团队配备内部 SRE 团队是不可能的，于是我的团队（SRE 咨询团队）诞生了。我们的目标很简单：推动这些没有 SRE 的团队实施高标准化。

虽然我们的任务很简单，但并没有明确的指示，所以我和我的团队就有了一定的自由空间来定义一系列标准，Uber 所有的微服务都可以遵循这些标准。从一开始就让这个庞大组织的每个微服务都遵循高标准不是一件容易的事情，于是在我的同事 Rick Boone（他的微服务高标准为这本书带来了启发）的帮助下，我创建了一个详细的标准检查列表。我相信，Uber 的每一个微服务在进入生产环境之前都应该遵循这些标准。

我们需要提炼出一系列原则，并提出具体的要求。最后我们提出了 8 项原则：Uber 的每个微服务都应该具备*稳定性*、*可靠性*、*伸缩性*、*容错能力*、*高性能*、*可监控*、*文档化*和*灾备能力*。每个原则都包含了具体的标准，这些标准对每个原则的具体含义进行了定义。重点是，我们要求每个原则都可以被量化，量化结果有助于提升微服务的可用性。如果一个微服务满足了这些标准和要求，我们就说它是*生产就绪*的。

如何在团队里高效地推行这些标准是接下来要做的事情。我建立了一个流程，对于关键性业务服务（这些服务的中断会拖垮整个应用），SRE 团队需要在团队间展开架构评审，收集审计反馈（关于每个服务是否满足生产就绪要求的检查列表），创建详细的路线图（把

微服务带向生产就绪状态的详细指南），并为每个服务的生产就绪程度打分。

架构评审是整个流程中最为重要的部分：所有相关的开发人员被聚集到一个房间里，他们被要求在 30 min 或更短的时间内在白板上画出服务的架构图。我的团队和开发人员可以快速地定位问题。当把微服务和所有相关元素（端点、请求消息流、依赖项等）都画在一起时，每一个故障点都会变得清晰可见。

架构评审卓有成效。每次评审之后，我们会核对检查列表，看看服务是否满足生产就绪要求，然后把结果分享给开发团队的经理和开发人员。我发现，在对服务进行生产就绪评估时，简单的生产就绪与否不足以准确地反映评估情况，所以我们加入了打分机制。每一项要求都对应一个分数，这些分数最后汇总成总分。

审计之后是创建路线图。路线图包含服务未能满足生产就绪要求的列表项，以及近期发生的中断情况、改进措施的描述、任务链接，以及相关开发人员的名字。

在做完这个流程（也被称为“Susan Flowler 的生产就绪流程即服务”）的生产就绪检查之后，下一步是对整个流程进行自动化，以便让 Uber 所有的微服务持续地执行这个流程。在写这本书的时候，无畏的 Roxana del Toro 正领导着他的 SRE 团队对整个流程进行自动化。

生产就绪标准里的每一项要求和实现细节都是我和我的 SRE 同事们经过无数个小时的细心工作才总结出来的。我们做了大量笔记，有过无数次的讨论，对微服务的方方面面（它们零零散散，有的领域甚至是一片空白）进行了全面调研。我与 Uber 和其他公司的微服务开发团队进行过交流，讨论如何对微服务进行标准化，看看是否存在一组标准原则可以应用在任意的微服务上，并对业务产生可量化的影响。这本书就是基于这些笔记、讨论、会议和调研而写的。

在与旧金山海湾地区其他公司的网站可靠性工程师和软件工程师进行交流之后，我才知道，在 SRE 领域，乃至整个技术行业，这都是一件非常有意思的事情。当有工程师向我询问微服务标准化和构建生产就绪微服务的相关问题时，我可以给他们提供建议，于是我写了这本书。

在写这本书的时候，关于微服务标准化的资料很少，关于如何构建和维护微服务生态系统的指南也很少，而当那些对单体应用进行微服务拆分的工程师问起“下一步我们该怎么办”时，更是没有一本书能够解答这个问题。我写本书的目的就是希望能够填补这些空白，并解答这个问题。当初我开始着手对 Uber 进行微服务标准化的时候是多么希望能有这样一本书啊。

这本书为谁而写

这本书主要是为微服务软件工程师和网站可靠性工程师而写的，他们要么苦于不知道该如何对单体系统进行分解，要么正在着手构建新的微服务，并希望能够构建出稳定的、可靠的、可伸缩的、容错的、高性能的微服务。

不过，书中所提及的相关原则不仅限于以上读者。大部分原则都可以被用于改进任何大小和任意架构的服务和应用。工程师、工程经理、产品经理和公司的高管都会从本书中获益，他们可以借此为他们的应用制订标准，从架构决策中理解组织结构的变更，或者把它们作为推动组织架构演变和运营的指南。

我假设读者对微服务的基本概念、微服务架构和现代分布式系统基础都有所了解，对于已经掌握了这些概念的读者来说，他们可以从书中获得更大的价值。对于还不熟悉这些概念的读者，我在本书的第 1 章专门对微服务架构、微服务生态系统、微服务给组织带来的挑战，以及将单体应用拆分成微服务的本质进行了描述。

如何定位这本书

这本书不是关于“如何做”的指南手册：它并没有为每一章所涉及的内容提供任何教程。如果要把它们写成教程，可以写出很多卷，因为每一章的内容都可以写成一本书。

所以，这是一本高度抽象的书，它具有很强的通用性，书中的内容几乎可以被应用于任何一家公司的任意一个微服务上。不过它也足够细致，工程组织可以把它作为切实可行的指南，用于改进和标准化微服务。每个公司的微服务生态系统都各不相同，遵循命令式或填鸭式的步骤指南没有任何意义。所以，我强调的是概念，解释了它们在构建生产就绪微服务方面起到的重要作用，并为每个概念提供示例和实现策略。

当然，这本书不是一本关于如何构建微服务和微服务生态系统的百科全书。首先，我得承认，构建微服务和微服务生态系统的方式有很多（例如，关于如何进行构建测试，除了在第 3 章里所提到的三阶段测试之外，还有其他很多方法），不过方法各有千秋，我尽可能向读者呈现最好的方法。

另外，技术的发展日新月异。如果有可能，我会尽量避免让读者局限在已有的技术上。例如，我在说明生产就绪日志的重要性时，并没有指定一定要使用 Kafka 来收集日志，而是让读者来选择特定的技术。

最后，这本书不是一本关于 Uber 工程组织的说明书。那些原则、标准、示例和策略并不只适用于 Uber，它们也不全是出自 Uber，它们的灵感来自很多科技公司，并且可以

被应用于任何微服务生态系统。它不是一部历史书籍，而是一部指导如何构建生产就绪微服务的指南。

如何使用这本书

读者可以根据具体需要来阅读这本书。

第一种是只阅读自己感兴趣的章节，其他章节可以略读，甚至不读。读者因此可以从书中获得如下好处：了解新的概念、获得对已有概念更深层次的理解、发现软件工程和微服务架构的新思维。

第二种是通读全书，并仔细阅读与自己的需求相关的章节，并把相关的原则和标准应用到实际中。例如，如果你需要对微服务的监控做出改进，那么就要仔细阅读第 6 章（监控），并根据这一章所提供的内容来改进微服务的监控、告警和中断响应流程。

最后一种是仔细阅读每个章节，理解书中所提到的每个标准和要求，并把它们应用到实际中。如果你的目标是对你的微服务或者整个公司的微服务进行标准化，让它们满足生产就绪标准，让微服务具备稳定性、可靠性、伸缩性、容错能力、高性能、监控能力、文档化和灾备能力，那么就应该采取这种方式进行阅读。

本书的第 3 章到第 7 章每一章会讲解一个微服务标准，在每一章的结尾部分都会有一个“评估你的微服务”小节，这个小节中列出了一个有关微服务的问题列表。这些问题是根据主题来分类的，你可以快速地从中找出感兴趣的问题，看看还需要做哪些事情来把你的微服务带向生产就绪状态。本书的结束部分有两个附录（附录 A 是关于生产就绪的检查列表，附录 B 是“评估你的微服务”的问题列表）可以用于追踪生产就绪标准，同时再次列出了所有“评估你的微服务”里所提到的问题。

本书的结构

第 1 章对微服务进行了介绍。它涵盖了微服务架构基础、将单体拆分成微服务的一些细节、微服务生态系统的 4 个层次，并有一个专门的章节描述了采用微服务架构的组织所面临的挑战和权衡。

第 2 章描述了对微服务进行标准化所要面临的挑战，并介绍了 8 个生产就绪标准，这些标准都是以微服务的可用性作为驱动的。

第 3 章描述了构建稳定可靠的微服务的原则，并介绍了开发周期、部署管道、依赖项的管理、路由和服务发现，以及如何稳定可靠地弃用微服务。

第 4 章介绍了构建具有伸缩性和高性能的微服务的要求，包括理清微服务的增长规模、有效地使用资源、了解资源、容量规划、依赖项伸缩、流量管理、任务处理，以及数据存储的伸缩。

第 5 章介绍了构建具有容错能力和灾备能力的微服务的原则，包括常见的故障场景、故障检测和修复策略、弹性测试，以及处理事故和中断的方法。

第 6 章介绍了微服务监控的详尽细节，以及如何通过标准化降低微服务监控的复杂性。这一章还介绍了日志、仪表盘和告警。

第 7 章介绍了微服务文档化以及用于促进开发团队和整个组织理解架构和运维的一些方法，并提供了一些实用的策略，用于在整个工程组织内实现生产就绪标准。

本书的结束部分包含两个附录。第 7 章的结束部分对附录 A“生产就绪检查列表”进行了描述，附录对分散在全书各处的生产就绪标准及其相应要求进行了简要的概括。附录 B“评估你的微服务”聚集了所有从第 3 章到第 7 章每章末尾列出的与“评估你的微服务”相关的问题。

本书使用的约定

本书使用的约定如下：

斜体（Italic）

用于表明新术语、网址、电子邮件地址、文件名和文件扩展名。

`等宽字体（Constant width）`

用于程序清单，以及在段落中对变量、函数名、数据库、数据类型、环境变量、语句和关键字等程序元素的引用。

`等宽加粗字体（Constant width bold）`

用于显示命令或其他需要用户输入的文字。

`等宽斜体字体（Constant width italic）`

用于显示应该由用户提供或者根据上下文确定的值。

该图标表示提示或建议。

该图标表示一般性的注释。

该图标表示提醒或警告。

中文版书中切口以“  ”表示原书页码，便于读者与英文原版图书对照阅读，本书的索引中所列的页码也为英文原版图书中的页码。

Safari Books Online

Safari（以前是 Safari Books Online）是一个会员制的为企业、政府、教育机构和个人提供培训和参考的平台。

会员可以通过几百家出版社的检索数据库访问几千种图书、培训视频和手稿。这些出版社包括 O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology 等。

更多关于 Safari Books Online 的信息，可以访问我们的网站 *http://oreilly.com/safari*。

如何联系我们

请将对本书的评价和存在的问题通过如下地址告知出版者。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网站，你可以在那里找到关于本书的相关信息，包括勘误列表、示例代码以及其他信息。本书的网站地址是：

http:/bit.ly/prod-ready_microservices

对于本书的评论和技术性的问题，请发送电子邮件到：

bookquestions@oreilly.com

关于我们的书籍、课程、会议和新闻的更多信息，请参阅我们的网站 *http://www.oreilly.com*。

在 Facebook 上找到我们：*http://facebook.com/oreilly*
在 Twitter 上关注我们：*http://twitter.com/oreillymedia*
在 YouTube 上观看我们：*http://www.youtube.com/oreillymedia*

致谢

谨以此书献给我的另一半 Chad Rigetti，他从繁忙的工作之中抽出时间聆听我关于微服务的见解，并时时给予我鼓励。如果没有他的支持，我很难完成此书。

感谢我的姐妹 Martha 和 Sara，她们勇敢、乐观、快乐，无时不刻在感染着我。还要感谢 Shalon Van Tine，她是我最亲密的朋友，多年来一直给予我莫大的支持。

感谢那些为本书的初稿提供反馈的人，感谢 Uber 的同事们，感谢那些勇于在他们的工程组织里应用微服务原则和策略的人。要特别感谢 Roxana del Toro, Patrick Schork, Rick Boone, Tyler Dixon, Jonah Horowitz, Ryan Rix, Katherine Hennes, Ingrid Avendano, Sean Hart, Shella Stephens, David Campbell, Jameson Lee, Jane Arc, Eamon Bisson-Donahue，和 Aimee Gonzalez。

如果没有得到 O'Reilly 工作人员的帮助，这本书不可能问世。所以要感谢技术审校 Brian Foster 和 Nan Barber，以及其他所有来自 O'Reilly 的工作人员。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- 提交勘误：你对书中内容的修改意见可在提交勘误处提交，若被采纳，将获赠博文视点社区积分（在你购买电子书时，积分可用来抵扣相应金额）。
- 交流互动：在页面下方读者评论处留下你的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/32433

第 1 章

微服务简介

在过去的几年，技术行业见证了行业巨头们（Netflix、Twitter、Amazon、eBay 和 Uber）在分布式系统架构上的快速转变，它们逐渐从单体架构转向微服务架构。微服务并非新生事物，现今那些基于微服务架构的应用程序，在一定程度上是为了解决伸缩性问题、运行效率问题和开发效率问题应运而生的。当复杂的软件系统被当作大型的单体应用进行部署时，引入新的技术变得非常困难，这也促使人们转向微服务架构。

使用微服务架构可以解决上述问题，不管是从一开始就使用微服务架构，还是把一个已有的单体应用拆分成可独立部署的微服务。使用微服务架构，应用程序可以在横向和纵向两个维度进行伸缩，开发速度也会得到大幅度提升，而且可以很容易地使用新技术替代旧技术。

我们将在这一章看到，使用微服务架构来构建具有伸缩性的应用程序是顺理成章的事。把一个单体应用拆分成微服务是出于伸缩性和效率方面的考虑，不过使用微服务本身也会引入一些新的挑战。成功的可伸缩微服务生态系统需要复杂且稳定的基础设施的支撑。另外，使用微服务架构的公司必须对组织结构进行重大调整，以便对微服务架构提供良好的支持，团队结构会因此出现无序的扩张。不过，微服务架构带来的最大挑战，是如何实现服务的标准化，以及如何保证服务的可靠性和可用性。

从单体应用到微服务

现今的每个应用程序几乎都可以被拆分为三个组件：*前端（客户端）*、*后端*和*数据存储*（参见图 1-1）。客户端向应用程序发送请求，后端负责处理复杂的逻辑，而那些需要被存储（基于内存或数据库）的数据被发送到可以存储数据的地方，或者从那里读取数据。我们把这叫作*三层架构*。

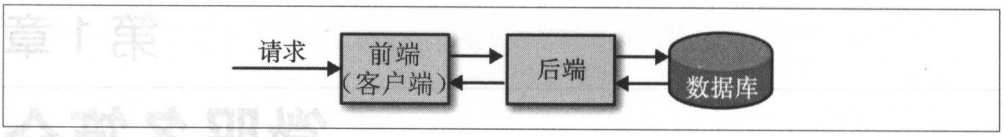


图1-1：三层架构

这三个组件可以通过三种方式来组合成一个应用程序。大多数应用程序把前面两个组件放在同一个代码库里，客户端和后端代码被视为一个独立的可执行文件，再加上一个独立的数据库。有的应用程序则把客户端和后端代码分开，它们都是独立的可执行文件，再加上一个外部数据库。而对于不需要外部数据库的应用程序，它们会把数据保存在内存里，所以会倾向于把三个组件都放在同一个代码库里。不过，不管这三个组件如何组合，我们总是把应用程序作为一个整体来看待。

应用程序从它们的生命周期开始就是按照这样的方式进行设计、构建和运行的，而且应用程序的架构一般与公司的业务和应用程序的功能相互独立。这三个组件作为应用程序组合的基础，会出现在每个网站、移动应用以及大型企业应用里。

在公司的早期发展阶段，应用程序比较简单，开发人员也不多，开发人员一般会分摊开发任务并维护同一个代码库。随着公司的发展，开发人员越来越多，新的功能不断地被添加到应用里，这时候会出现三个问题。

首先，运维工作量会增加。简单地说，运维工作就是那些与应用程序的运行和维护相关的工作。公司需要雇佣专门的运维工程师（系统管理员、技术支持工程师和所谓的DevOps 工程师），他们会做一些跟硬件、监控相关的事情，并且要随时待命。

3 对于第二个问题，我们可以用简单的数学运算来说明：往应用程序里添加新功能，会增加代码行数和应用程序本身的复杂性。

第三个问题，添加新功能催生了对应用程序进行横向或纵向扩展的需求。业务流量的增长对应用程序的伸缩性和性能提出了更高的要求，需要在更多的服务器上部署应用程序。于是更多的服务器被添加进来，负载均衡器把请求均衡地分布到各个服务器上（如图1-2 所示，包括一个前端组件，中间是负载均衡层，后面是服务器）。纵向扩展也变得很有必要，因为功能越多，应用程序需要处理的任务也越多，它们需要被部署在能够承担更多 CPU 计算任务和拥有更大内存的服务器上（参见图 1-3）。

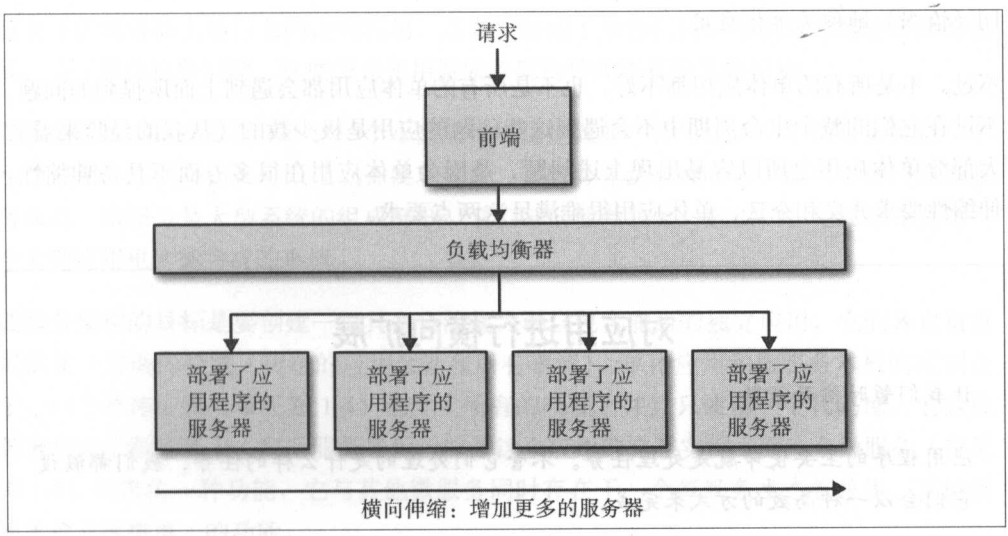


图1-2：横向伸缩一个应用

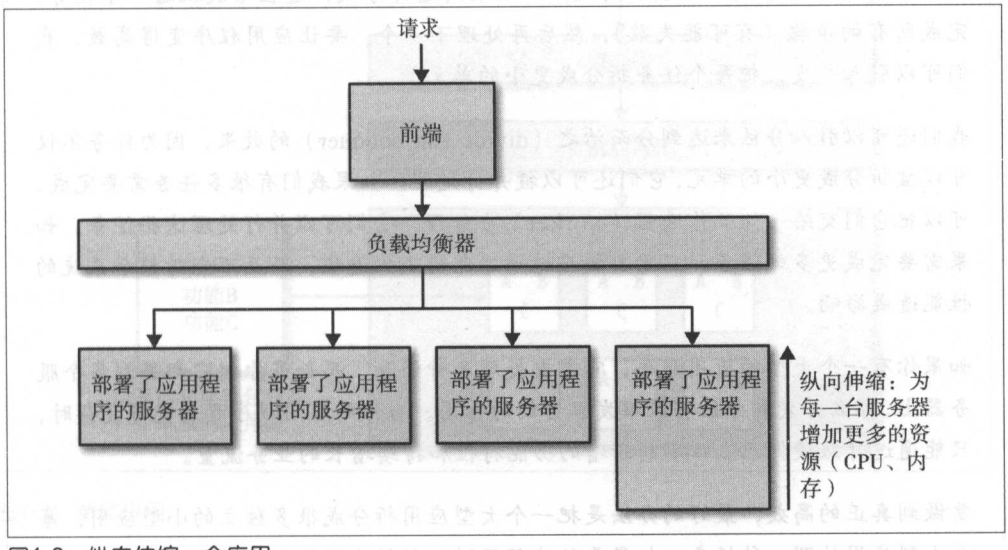


图1-3：纵向伸缩一个应用

随着公司的发展，开发人员的数量可能超过 3 位数，事情开始变得复杂起来。开发人员 4
不断地往代码库里添加新功能、补丁和修复缺陷的代码，应用程序的代码行数因此达到了数以万计的规模。应用程序的复杂性在持续增加，为了确保代码变更（哪怕只修改了一两行代码）不破坏已有代码的完整性，数以百计（如果不是数以千计）的测试用例被添加进来。开发和部署会成为噩梦，测试会成为发布缺陷修复的负担和阻碍，技术债务成堆地出现。软件社区把生命周期符合这种模式（比这更好或比这更糟）的应用程序亲

切（恰当）地称为单体应用。

不过，不是所有的单体应用都不好，也不是所有的单体应用都会遇到上面所提到的问题，不过在它们的整个生命周期中不会遇到这些问题的应用是极少数的（从我的经验来看）。大部分单体应用之所以容易出现上述问题，是因为单体应用在很多方面不具备伸缩性。伸缩性要求并发和分区，单体应用很难满足这两点要求。

5

对应用进行横向扩展

让我们暂时岔开话题。

应用程序的主要使命就是处理任务。不管它们处理的是什么样的任务，我们都假设它们会以一种高效的方式来完成。

并发是高效处理任务的一种方式。也就是说，我们不能只用一个进程来处理所有的任务，因为这种方式一点也不高效！如果采用这种方式，进程每次挑选一个任务，完成所有的步骤（有可能失败），然后再处理下一个。要让应用程序变得高效，我们可以引入并发，把每个任务拆分成更小的单元。

我们还可以引入分区来达到分而治之（divide and conquer）的效果，因为任务不仅可以被拆分成更小的单元，它们还可以被并行处理。如果我们有很多任务需要完成，可以把它们交给一组工作进程（worker）去处理，它们可以并行处理这些任务。如果需要完成更多的任务，只需要相应地增加新的工作进程，而且不会对整个系统的性能造成影响。

如果你有一个大型的应用程序，它需要处理各种任务，而如果你把它部署到多个服务器上，那么并发和分区将很难发挥它们的作用。当你的应用程序开始变得复杂时，只能通过升级硬件来支撑不断新增的功能特性和持续增长的业务流量。

要做到真正的高效，最好的办法是把一个大型应用拆分成很多独立的小型应用，每个小型应用处理一种任务。如果要往流程里增加新的步骤也很简单，只需要增加一个新的小型应用来处理这个步骤！如果要处理更多的业务流量也很简单，在每个小型应用里增加更多的工作进程！

单体应用很难支持并发和分区，这直接导致了单体架构在效率方面无法达到我们的期望。

我们在 Amazon、Twitter、Netflix、eBay 和 Uber 这样的公司里看到了这种模式——在成

百上千的服务器上运行着的这些应用，逐渐发展成了单体应用，并面临伸缩性方面的挑战。为了解决这些问题，这些公司使用微服务架构替代原有的单体架构。

微服务的基本原理很简单：让一个小型应用专注地做好一件事情。一个微服务就是一个小型应用，方便替换，可以独立开发和部署。微服务无法单独存在，所以不会出现微服务孤岛。微服务是大型系统的组成部分，它们与其他微服务一起工作，完成原先要在一个大型应用里才能完成的事情。 6

微服务架构的目标是要创建一组具备自治能力和自包含能力的独立应用，它们各自负责提供某一方面的功能（传统的应用会处理所有事情）。单体应用和微服务本质的区别在于，一个单体应用（参见图 1-4）包含了所有的功能，并且只使用一个代码库，它会被部署到多个服务器上，每个服务器运行的是这个应用的完整实例。而一个微服务（参见图 1-5）只提供一种功能，它与其他微服务同时存在于一个微服务生态系统里，其他微服务也只提供单一的功能。

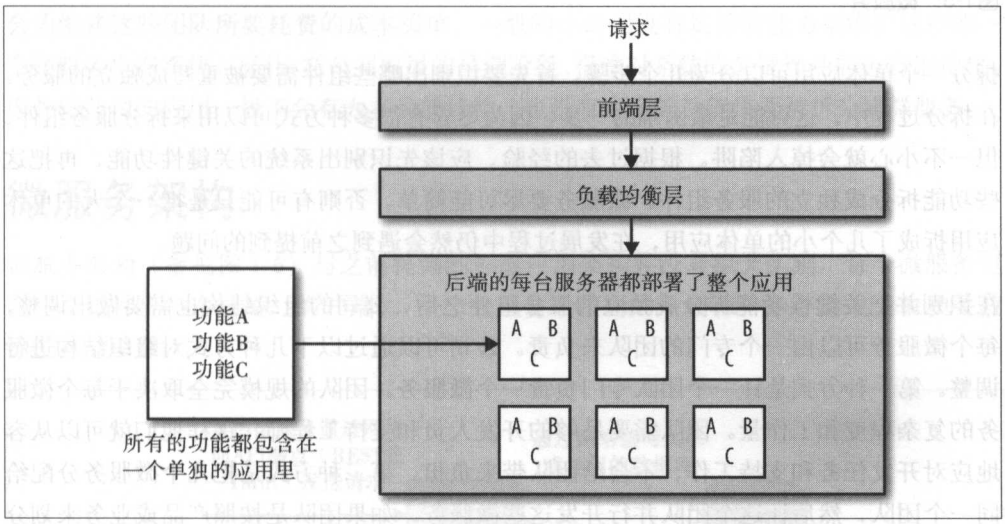


图1-4: 单体应用

使用微服务架构有很多好处，它可以减少技术债务，提升开发和测试效率，提升伸缩性，简化部署，等等。转向使用微服务架构的公司一般都是因为之前构建的应用达到了伸缩性的瓶颈。他们最开始构建的是单体应用，然后把单体应用拆分成微服务。 7

把单体应用拆分成微服务的难易程度取决于单体应用本身的复杂程度。成功拆分一个包含大量特性的单体应用需要花费很多精力，而且要深思熟虑，它还会涉及团队结构的重组。所以，转向微服务架构需要整个公司一起努力。

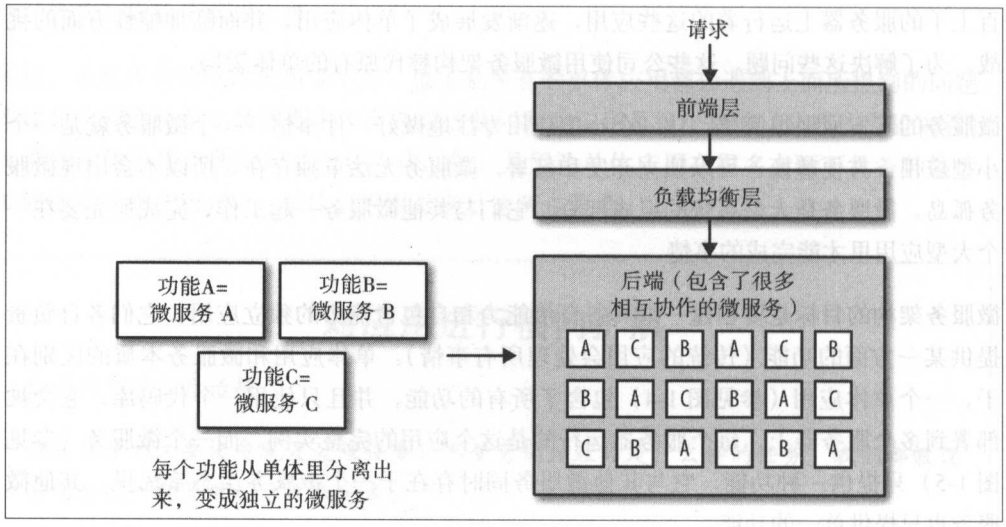


图1-5：微服务

拆分一个单体应用可以分为几个步骤。首先要识别出哪些组件需要被重写成独立的服务。在拆分过程中，这可能是最困难的一步，因为尽管有很多种方式可以用来拆分服务组件，但一不小心就会掉入陷阱。根据过去的经验，应该先识别出系统的关键性功能，再把这些功能拆分成独立的服务组件。微服务要尽可能简单，否则有可能只是把一个大的单体应用拆成了几个小的单体应用，在发展过程中仍然会遇到之前提到的问题。

在识别并把关键性功能拆分成独立的服务组件之后，公司的组织结构也需要做出调整，
8 每个微服务可以由一个专门的团队来负责。公司可以通过以下几种方式对组织结构进行调整。第一种方式是让一个团队专门负责一个微服务。团队的规模完全取决于每个微服务的复杂程度和工作量。团队需要足够的开发人员和支持工程师，这样他们就可以从容地应对开发任务和支持工作，不会给团队带来负担。第二种方式是把几个微服务分配给同一个团队，然后让这个团队并行开发这些微服务。如果团队是按照产品或业务来划分的，那么这种方式是最好的，因为与该产品或业务相关的服务都可以由同一个团队负责开发。如果选择了第二种方式，那么要确保不要让开发人员承担超负荷的工作量，也不要让他们太过疲惫。

创建微服务生态系统是微服务架构另一个重要的组成部分。一般来说（或者说至少），一个运行着大型单体应用的公司都会有一个专门的团队，这个团队负责设计、构建和维护可供这个单体应用运行的基础设施。在对这个单体应用进行拆分时，这个团队需要为微服务提供一个稳定的运行环境，他们的职责会变得相当重要。他们需要为开发团队提供稳定的基础设施，并隐藏微服务交互的复杂性。

在完成了应用组件化、团队结构调整和基础设施团队建设之后，整个迁移过程就可以开始了。一些团队选择从整个代码库里把与特定微服务相关的代码拉取出来，用它们构建独立的微服务。在确保这些微服务可以独立承担相关任务之前，它们会跟原先的单体应用共存一段时间。有些团队则选择重新构建微服务，使用全新的代码库，在经过适当的测试之后其就可以承担起处理业务流量的工作。选择哪一种方式取决于微服务的功能特性，不过据我所知，这两种方式在大多数情况下都可以很好地工作。周密的计划和强大的执行力是决定一个迁移能否成功进行的关键所在，而且对一个大型的单体系统进行彻底迁移可能需要花上几年时间。

从一开始就使用微服务架构比迁移一个单体应用到微服务看起来要好得多，毕竟它不需要面对伸缩性方面的挑战，也不需要做任何迁移。对于一些公司来说也许是这样的，不过对此我要多说几句。小公司一般没有符合要求的基础设施来运行微服务，哪怕是很小规模的微服务。好的微服务架构需要稳定的基础设施，这样的基础设施一般都很复杂，需要专门的团队来负责运维。只有在面临伸缩性挑战并决定要转向微服务架构的公司才会为组建这些团队所要耗费的成本买单，一般的小公司没有足够的能力来维护这样的一个微服务生态系统。况且，在公司发展的早期阶段，很难对系统的关键性功能进行组件化，因为新公司的应用一般不会有太多功能特性，也没有太多的功能需要被拆分成微服务。

微服务架构

微服务架构（参见图 1-6）与之前提到的标准应用架构并没有很大区别。每个微服务包含三个组件：一个前端（客户端）、一个处理复杂逻辑的后端和一个存储或读取相关数据的存储引擎。

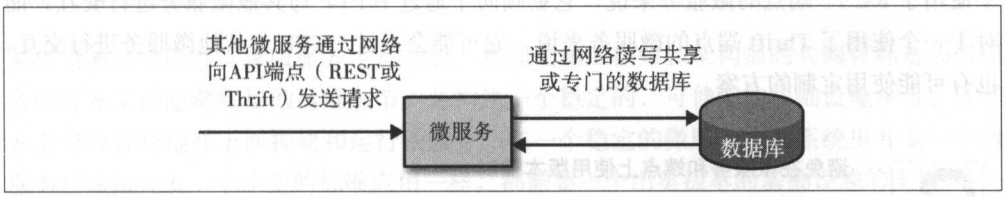


图1-6：微服务架构元素

这里所说的前端，并不是指一般的前端应用，而是一套包含了固定端点的 API。设计良好的微服务 API 之间可以进行简单且高效的交互，只需把请求发送到相应的端点上。例如，一个负责处理客户数据的微服务可能会有一个 *get_customer_information* 端点，其他服务可以将请求发送到这个端点上，以便获得客户的相关信息。还有一个 *update_customer_information* 端点，其他服务可以发送请求到这个端点上，以便更新客户信息。以及一个 *delete_customer_information* 端点，其他服务可以使用这个端点删除客户信息。

从理论上说，这些端点可以被单独分离出来，但实际上不能这样做，因为它们都是作为整个后端的组成部分而存在的。这些微服务负责处理客户数据，一个发送到 *get_customer_information* 端点的请求将会触发一系列任务，包括处理发送过来的请求、调用过滤器、从数据库里读取数据、对数据进行格式化，并把数据返回给客户端（其他微服务）。

10 大部分微服务会存储数据，不管是存在内存里（比如缓存）还是存在外部数据库里。如果数据是被保存在内存里，那么就没必要通过网络调用外部数据库，微服务自己就可以很容易地把数据返回给客户端。如果数据是被保存在外部数据库里，那么微服务就需要向数据库发起请求，在收到响应之后继续处理其他任务。

在需要让微服务相互协作的情况下，这种架构是很有必要的。由一组相互协作的微服务组成一个生态系统，这是微服务架构规范的基本要求，否则就又回到单体架构上了。所以，如果要让一组微服务进行高效的交互，需要在整个组织内对微服务架构的几个元素进行标准化。

微服务 API 的端点需要在组织内进行标准化。并不是说所有微服务都必须有固定的端点，而是说端点的类型应该保持不变。REST 和 Apache Thrift 是两种很常见的 API 端点类型，我见过有些微服务同时使用这两种端点（不过这种情况很少见，它会让监控变得复杂，所以我不建议这样做）。选择哪种端点反映了微服务内部的工作机制，同时也说明了架构情况。例如，异步微服务无法通过基于 HTTP 的 REST 端点进行交互，它要求使用基于消息队列的端点。

微服务可以通过*远程过程调用*（RPC）进行交互，RPC 让跨越网络的调用看起来跟本地调用一样。是否使用 RPC 取决于架构的选择、组织的支持和端点的类型。例如，对于一个使用了 REST 端点的微服务来说，它更倾向于通过 HTTP 与其他微服务进行交互，而对于一个使用了 Thrift 端点的微服务来说，它可能会通过 HTTP 与其他微服务进行交互，也有可能使用定制的方案。

避免在微服务和端点上使用版本控制

微服务并不是在编译时或运行时被加载到内存的软件包，它们是独立运行的应用程序。微服务的演化速度很快，如果对它们进行版本控制管理，客户端的开发人员需要在不同的微服务版本（过时或无人维护）间进行切换，这对组织来说会是一场噩梦。微服务会持续变化，并不是固定发布的软件包。同样，对 API 端点进行版本控制管理也是一种需要避免的反模式。

11 任何用来作为微服务交互手段的端点和协议都具有两面性。使用哪种端点或协议不应该由某个微服务开发者来决定，它应该是微服务生态系统整体架构设计的一部分（后面我们会详细说明）。

微服务为开发者提供了很大的自由空间，除了由组织选定的API端点类型和通信协议之外，开发者可以按照自己的想法来实现微服务的内部结构。只要遵循选定的端点类型和通信协议，他们可以使用任何一种编程语言，比如Go、Java、Erlang或者Haskell。开发一个微服务与开发一个标准应用并没有很大的不同。不过有一个需要注意的地方，开发者对语言选择的自由度对一个工程组织来说将会变成一个巨大的成本，我们将在这一章的结束部分（“组织的挑战”一节）说明这一点。

这么说来，其他人会把微服务看成一个黑匣子：你通过向它的一个端点发送请求来输入信息，然后得到一些输出。如果你在一个合理的时间内获得你想要的东西，而且没有出现任何错误，那么可以说微服务很好地完成了它的工作，因此我们并没有必要知道除端点之外的任何事情，也不需要知道服务是否工作正常。

关于微服务架构具体细节的讨论在这里要告一段落，不过这些并不是微服务架构的全部，后面的每个章节都会对微服务的各个方面进行详细的探讨。

微服务生态系统

微服务并不是孤立存在的，它们存在于一个环境里，微服务在这个环境里进行交互。大型微服务环境的复杂性跟热带雨林、沙漠或大洋的生态复杂性有得一比。把这种环境看成*微服务生态系统*，有助于理解微服务架构。

在一个设计良好的微服务生态系统里，微服务与基础设施之间是分离的。微服务与硬件、网络、构建和部署管道、服务发现和负载均衡都是分离的。它们都是微服务生态系统基础设施的组成部分。如何以一种稳定可靠的、可伸缩的、可容错的方式来构建、维护和标准化基础设施，是微服务运维的根本。

基础设施必须能够支撑微服务生态系统。基础设施工程师和架构师的共同目标是为微服务的开发工作隐藏底层的运维细节，并构建一个稳定的、可伸缩的基础设施平台，让开发者可以轻松地在上面构建和运行微服务。在一个稳定的微服务生态系统里开发一个微服务应该跟开发一个小型的标准应用一样，都需要一个出类拔萃的基础设施的支持。

微服务生态系统可以被分为4层（参见图1-7），虽然层与层之间的边界不一定都很清晰，但这些层会涉及基础设施的几个元素。底下的3层是基础设施层：最下面的是硬件层，其次是通信层（一直渗透到第4层），紧接着的是应用平台层。第4层（顶层）就是微服务所在的层。

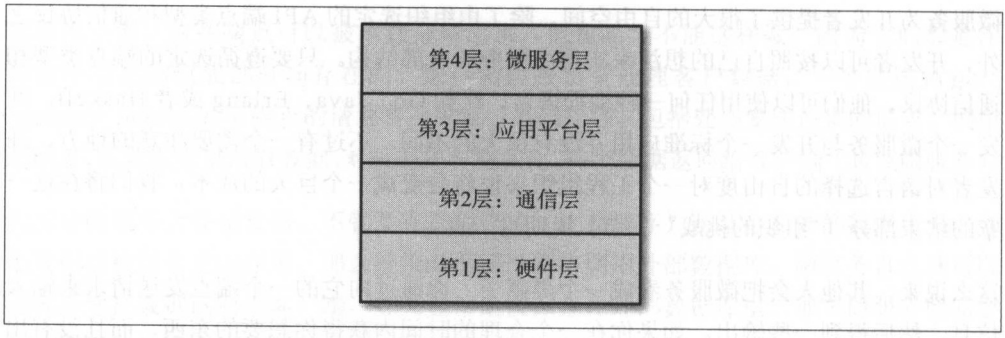


图1-7：微服务生态系统的4层模型

第 1 层：硬件层

微服务生态系统的底层是*硬件层*。这一层是服务器物理机所在的层，它们是所有内部工具和微服务运行的基础。这些服务器被放置在数据中心的机架上，由供电系统供给电力，使用着昂贵的 HVAC 冷却系统。这里有各种各样的服务器：有些专门用于运行数据库，有些专门处理 CPU 密集的任务。它们有些是某些公司私有的，有些是从所谓的“云服务提供商”那里租来的，比如 Amazon Web Services Elastic Compute Cloud（AWS EC2）、Google Cloud Platform（GCP）或 Microsoft Azure。

硬件的选择取决于服务器的所有者。如果你的公司有自己的数据中心，硬件的选择就是自己的事情，完全可以根据实际需要来定制服务器。如果你的服务器是在云端（这种情
13 况比较普遍），你就没有太多的选择余地。是*自己购买服务器还是选择云服务*并不是一个简单的选择题，它需要考虑购买成本、可用性、可靠性和运营成本。

管理服务器是硬件层的职责之一。每台服务器都需要安装标准的*操作系统*。使用哪种操作系统并没有一个标准的答案，这完全取决于你要构建的应用程序、构建应用程序所使用的编程语言以及构建微服务所需要的软件包和工具。主流的微服务生态系统一般会使用 Linux 的变形版本，比如 CentOS、Debian 或 Ubuntu，不过一个使用了 .NET 平台的公司显然会有不同的选择。硬件层之上还可以有一些层，比如资源隔离、资源抽象（由 Docker 和 Apache Mesos 提供的）和数据库（专门的或共享的）。

操作系统的安装和硬件资源的配置是服务器的第一个层。每个主机必须被配置好，而且在安装好操作系统之后，必须提供一个*配置管理工具*（比如 Ansible、Chef 或 Puppet）来安装应用程序，并做好必要的配置。

对主机进行*主机级别的监控*（使用 Nagios）是有必要的，而且需要*记录主机级别的日志*。在主机出现异常（磁盘错误、网络错误或 CPU 过载）时就可以方便地对它们进行诊断，

有助于问题的解决。主机级别的监控将在第 6 章进行详细描述。

硬件层的主要内容

微服务生态系统的硬件层（第 1 层）包含：

- 物理服务器（公司自有或从云服务提供商那里租用）
- 数据库（专有的或共享的）
- 操作系统
- 资源隔离和资源抽象
- 配置管理
- 主机级别的监控
- 主机级别的日志

第 2 层：通信层

微服务生态系统的第 2 层是通信层。通信层渗透到生态系统的所有层（包括应用平台层和微服务层），因为微服务之间的交互会在多个层上进行，所以很难清晰地对通信层与其他层之间的边界进行定义。虽然难以清晰地定义它们之间的边界，但是这个层所涉及的元素是很明确的。第 2 层一般包含网络、DNS、RPC 和 API 端点、服务发现、服务注册以及负载均衡。

有关通信层网络和 DNS 元素的内容已经超出了本书的范畴，在这一小节，我们主要讨论 RPC、API 端点、服务发现、服务注册和负载均衡。

RPC、端点和消息传递

微服务通过远程过程调用（RPC）或消息传送与其他微服务进行交互，这些调用通过网络发送到其他微服务的 API 端点上（如果使用的是消息传递，消息会被发送到消息代理，消息代理会对这些消息进行路由）。基本原理是这样的：使用一个特定的协议，一个微服务把符合特定格式的数据通过网络发送到另一个服务（或者是另一个微服务的 API 端点）或消息代理上（消息代理确保数据会被路由到其他微服务的 API 端点上）。

微服务有几种通信方式，第一种是最常用的 HTTP+REST/Thrift。如果使用的是这种方式，各个服务使用超文本传输协议（HTTP）进行网络交互，它们向特定的 REST 端点（使用各种 HTTP 方法，比如 GET、POST 等）或 Thrift 端点发送请求或从这些端点接收响应。发送的数据一般是 JSON（或 protocol buffer）格式。

HTTP+REST 是最便利的微服务通信方式。它使用起来很简单，而且稳定可靠。不过它的不足之处在于它是同步（阻塞）的。

第二种通信方式是*消息传递*。消息传递是异步（非阻塞）的，不过相对复杂。消息传递的工作原理是这样的：一个微服务把数据（消息）通过网络（HTTP 或其他）发送给一个消息代理，消息代理会把消息路由到其他微服务上。

消息传递也有几种模式，最流行的两种分别是发布和订阅以及请求和响应。如果使用的是发布和订阅模式，客户端会订阅一个主题，它将从主题上收到发布者发布的任何一个消息。请求和响应模式就更直接了，客户端发送一个请求到一个服务（或消息代理）上，这个服务会对这个请求做出响应。有些消息中间件同时支持两种模式，比如 Apache Kafka。Celery 和 Redis（或 Celery 和 RabbitMQ）可以为 Python 微服务传送消息（并处理任务）：Celery 处理任务或消息，Redis 或 RabbitMQ 作为消息代理。

消息传递有几个缺点需要注意。消息传递不会比 HTTP+REST 具备更强的伸缩性，如果你的系统对伸缩性有要求的话一定要清楚这一点。消息传递对变更不友好，因为它是集中式的，这样会导致消息队列和消息代理变成整个生态系统的故障点。它的异步特性在并发环境里会导致竞赛条件，如果没有处理好，还会出现无限循环。在使用消息传递时，如果能够处理好上述这些问题，它会变得跟同步解决方案同样稳定和高效。

服务发现、服务注册和负载均衡

在单体应用架构里，所有的业务流量都被发送给负载均衡器，然后被分发到应用服务器上。而在微服务架构里，业务流量被路由到大量不同的应用程序上，然后再被分发给部署了特定微服务的服务器。为了能够高效地实现上述场景，微服务架构需要在通信层实现三项技术：*服务发现*、*服务注册*和*负载均衡*。

一般来说，如果微服务 A 需要向微服务 B 发起请求，那么微服务 A 需要知道微服务 B 的 IP 地址和端口。微服务的通信层需要知道这些微服务的 IP 地址和端口，才能正确地路由这些请求。这个问题可以通过*服务发现*（比如 etcd、Consul、Hyperbahn 或 ZooKeeper）来解决，服务发现可以确保请求会被路由到它们本该去的地方，而且只会被路由到正常运行的实例上（这非常重要）。服务发现需要用到*服务注册*，服务注册中记录了生态系统里所有微服务的 IP 地址和端口。

动态伸缩和端口分配

在微服务架构里，在对微服务进行横向扩展和重新部署时（比如使用了像 Apache Mesos 这样的硬件抽象层），端口和 IP 地址会发生变化。在这种情况下，可以考虑为每个微服务分配一个静态端口（包括前端和后端）。

除非你的所有微服务都部署在同一个实例上（一般不太可能），否则需要在通信层使用负载均衡。简单地说，负载均衡可以做到：如果你有 10 个微服务实例，负载均衡器（软件或硬件）可以确保业务流量被（均衡地）分发到所有的实例上。在微服务生态系统里，只要涉及请求转发，都需要用到负载均衡器，这意味着一个大型的微服务生态系统将包含多层负载均衡。常见的负载均衡器有 Amazon Web Services Elastic Load Balancer、Netflix 的 Eureka、HAProxy 和 Nginx。

16

通信层的主要内容

微服务生态系统的通信层（第 2 层）包含：

- 网络
- DNS
- 远程过程调用（RPC）
- 端点
- 消息传递
- 服务发现
- 服务注册
- 负载均衡

第 3 层：应用平台层

应用平台层是微服务生态系统的第 3 层，这一层包含了所有独立于微服务的内部工具和服务。这一层所包含的集中式的工具和服务跨越了整个生态系统，因为有了这些工具和服务，微服务开发团队就可以把精力集中在微服务的开发上。

一个好的应用平台需要为开发者提供一套内部的自助工具，包括标准化的开发流程、集中式的自动化构建和发布系统、自动化测试、标准化和集中式的部署方案以及集中式的日志和微服务级别的监控。这些元素的细节将在后续章节中进行探讨，不过我们也会在这一章简要地介绍其中的几个元素，阐述一些基本的概念。

内部自助开发工具

17

有很多东西可以被纳入内部自助开发工具的范畴，它们是否可以被归为这类工具不仅取决于开发者对工具的需求，还要考虑基础设施和生态系统的整体抽象度和复杂性。决定使用哪一种工具，首先要对责任领域进行切分，然后对开发者所要完成的任务进行评估，以便设计、构建和维护他们的服务。

在一个已经使用了微服务架构的公司里，给工程团队指派职责要十分谨慎。最简单的做

法是为微服务生态系统的每一个层组建一个工程子团队。这些工程子团队将负责处理它们所在层的所有相关事务：运维团队负责第 1 层，基础设施团队负责第 2 层，应用平台团队负责第 3 层，微服务团队负责第 4 层（这看起来很简单，只要你能明白就好了）。

在这种组织结构里，工作在上层的工程师需要使用自助工具对下层的一些东西进行配置。例如，负责消息服务的团队应该为其他开发者提供一个自助工具，当微服务团队的开发者需要为他们的服务配置消息系统时，他们就可以使用这个工具，而无须过多地了解纷繁复杂的消息系统。

使用这些集中式的自助工具是有原因的。在一个多元化的微服务生态系统里，一个团队的普通工程师对其他团队的系统和服务并不了解（或知之甚少），他们也不可能成为面面俱到的专家。每个开发人员只对自己负责的部分比较了解，但从整个生态系统来看，这些开发人员组合在一起就无所不知了。为生态系统的每一部分构建易用的用户界面，为开发人员提供相关的培训，以便教会他们如何使用这些工具，而不是试图让每个开发人员了解这些工具和服务纷繁复杂的内部细节。把所有的事情放到一个黑匣子里，然后提供详细的说明文档。

使用这些工具的第二个理由是，你不需要其他团队的人来对你的服务和系统做任何关键性的改动，因为这些人可能会给你们带来麻烦。对于底层（第 1 层、第 2 层和第 3 层）的服务和系统来说，更是如此。让他们在这些层上面做出改动，或者要求（更糟糕的是
18 期待）他们成为某方面的专家有可能会酿成大祸。举一个配置管理的例子：不具备相关专门知识的微服务团队开发人员对系统配置做了一些变更，这有可能导致大规模的服务瘫痪，因为他们所做的变更有可能不只是影响到他们自己的服务。

开发周期

开发人员在对已有微服务进行修改或构建新的微服务时，对开发流程进行流水线化、标准化和自动化可以大幅提升开发效率。对开发流程进行标准化将在第 4 章进行探讨。有些东西需要被放在微服务生态系统的第 3 层，让稳定可靠的开发成为可能。

首先是集中式的版本控制系统，这个系统保存了所有代码，允许对代码进行跟踪、版本管理和搜索。这个可以通过一些工具来实现，比如 GitHub 或者自有的 git 或 svn 代码仓库，可以将这些仓库和一些协作工具集成起来，比如 Phabrictor，以简化代码的维护和审查工作。

其次是稳定高效的开发环境。众所周知，在微服务生态系统里实现一个这样的开发环境是很困难的，因为微服务之间的依赖太过复杂。不过它们都是最基本的因素，我们无法避免。一些工程组织倾向于在本地完成开发工作（在开发人员的电脑上），不过这样会导致糟糕的部署，因为开发人员并不清楚他们修改的代码是如何被部署到生产环境的。最稳定可靠的构建开发环境的方式是为生产环境创建一个镜像（不是为了预生产，也不是为了收集反馈，更不是为了生产），这个镜像包含所有复杂的依赖关系链。

测试、构建、打包和发布

开发过程中的测试、构建、打包和发布应该尽量被标准化和集中化。在开发结束之后，当有代码变更被提交，需要运行相关的测试用例，然后自动构建和打包即将发布的新版本。这个时候，持续集成工具就可以派上用场，一些现成的解决方案（比如 Jenkins）不仅功能齐全而且使用方便。这些工具可以让整个过程自动化，几乎不留给人类任何犯错的机会。

部署管道

在经过了开发、测试、构建、打包和发布这些步骤之后，部署管道是新代码走向生产环境的另一个流程。在一个微服务生态系统里，部署会在很短的时间内变得极其复杂，每天上百个部署都是很平常的事。开发团队需要为开发构建工具，并对开发过程进行标准化。关于如何构建稳定可靠（为生产环境准备的）的部署管道将在第 3 章进行探讨。

日志和监控

所有的微服务都应该把与它们的请求和响应相关的重要信息记录到日志里。因为微服务变更的速度太快，如果系统发生了错误，重建当时的系统状态变得很困难，导致代码的缺陷难以重现。使用微服务级别的日志可以帮助开发人员更好地了解他们的服务在过去某个时刻或当前时刻的状态。在微服务级别对微服务的关键度量指标进行监控也是出于同样的目的：实时准确的监控可以帮助开发人员了解服务的状态和健康状况。微服务级别的日志和监控将在第 6 章进行详细探讨。

应用平台层的主要内容

微服务生态系统的应用平台层（第 3 层）包含：

- 内部自助开发工具
- 开发环境
- 测试、构建、打包和发布工具
- 部署管道
- 微服务级别的日志
- 微服务级别的监控

第 4 层：微服务层

微服务生态系统的顶层是微服务层（第 4 层）。这一层是微服务以及微服务所有相关事物所在的层，它与底下的基础设施层完全分离，比如硬件、部署、服务发现、负载均衡

20 和通信。微服务层唯一没有被分离的是使用自助工具所做的配置。

在软件工程里，应用的配置一般会被集中化，针对某个工具或某些工具（配置管理、资源隔离或部署工具）的配置可以和这些工具保存在一起。例如，应用程序的自定义部署配置一般会和部署工具的代码保存在一起，而不是和应用程序的代码保存在一起。这种方式对单体应用架构或小型的微服务生态系统来说是没有问题的，但在包含了大量微服务和内部工具（每个工具都有自定义的配置）的大型微服务生态系统里，这种方式就会造成混乱：处在上层的微服务团队需要修改处在下层的工具代码，他们会经常忘记哪些地方包含了配置信息（或者不包含）。为了解决这个问题，可以把与微服务相关的配置放在微服务代码库里，然后开放给下层的工具和系统访问。

微服务层的主要内容

微服务生态系统的微服务层（第4层）包含：

- 微服务
- 微服务相关的配置

组织的挑战

微服务架构解决了在单体架构中存在的最棘手的问题。微服务不会被伸缩性、效率或技术更新能力方面的问题所困扰。相反，微服务在伸缩性、效率和开发速度上具有显著的优势。在一个技术日新月异的行业，尝试维护和改进一个笨重的单体应用不是一个明智之举。仔细想想，为什么会有人愿意勉强自己把一个单体拆分成微服务？为什么会有人愿意煞费苦心地从头创建一个微服务生态系统？

微服务看起来就像一个神奇（某种程度上说确实是这样）的解决方案，不过我们应该更清楚它的底细。在《人月神话》这本书里，Frederick Brooks 解释了为什么软件工程里不
21 存在银弹，他说："不管在技术还是管理领域，都不存在这样的一种承诺，它能够在一段时期内为生产力、可靠性和简单性带来哪怕是一个数量级的增长。"

当我们认为某项技术将为我们带来戏剧性的转变时，我们要仔细斟酌。微服务向我们承诺更好的伸缩性和更高的效率，但我们知道，这需要在整个系统的某些部分付出一些代价。

我们需要仔细权衡微服务架构带来的4个挑战。首先，组织结构需要做出调整，这会弱化跨团队之间的沟通，而且违反了康威定律。其次，技术的蔓延无法预测，这种蔓延对

于整个组织和每个工程师来说都是巨大的成本。第三，系统失效的可能性会因此增加。第四，工程资源和基础设施资源的争夺会变得激烈起来。

反康威定律

康威定律（1968 年，Melvin Conway 提出该定律，并以他的名字命名）的基本思想是：公司的沟通方式和组织结构决定了系统的架构。把康威定律反过来说（我们把它称为反康威定律）也是成立的，而且它与微服务生态系统是相契合的：产品的架构决定了公司的组织结构。在康威定律提出后的四十多年来，康威定律和反康威定律依然奏效。如果把微软的组织结构图画出来，它看起来跟微软的产品架构很像。谷歌、亚马逊，以及其他大型的科技公司也是这样的。使用了微服务架构的公司毫无例外地遵循着这个定律。

微服务架构由大量小型独立的微服务组成。根据反康威定律，对于那些使用了微服务架构的公司来说，它们都是由很多小型的独立团队组成的。团队结构会不可避免地出现无序的扩张，而且当微服务生态系统变得越来越复杂，要求更大的并发量和更高的效率时，这个问题会变得越来越突出。

反康威定律在某种程度上也意味着开发者就像微服务一样：他们专注做一件事情，而且可以把它做到最好（但愿如此），他们可以独立于（在职责、领域知识和经验方面）生态系统里的其他部分。如果从整体来看，把微服务生态系统的所有开发人员组合在一起，可以说他们是无所不知的。但如果从个体来看，他们只知道他们所负责的整个生态系统的一小部分。

这就出现了一个不可避免的组织问题：虽然微服务必须由单独的团队来开发（导致出现大量独立的团队），但他们并不是孤立的，他们必须通过无缝的沟通来完成整个产品的功能。这些独立的功能团队必须进行紧密的合作，而且团队的目标和项目要尽量与团队所开发的微服务相吻合（做到这点有点困难）。 22

在微服务团队和基础设施团队之间也有一些沟通上的问题需要解决。例如，应用平台团队需要为微服务团队构建他们需要的服务和工具，但光是从数百个微服务团队那里收集需求就需要花上数月（甚至数年）的时间。让开发者团队和基础设施团队在一起工作并不是一件容易的事情。

反康威定律也会带来一个问题，这个问题在使用单体架构的公司里很少会出现：组建运维团队会变得很困难。对于一个单体应用来说，运维团队直接为这个应用负责。但在微服务架构里就很难做到这点，因为每个微服务需要由单独的开发团队和运维团队来负责。这样会导致开发团队在开发微服务的同时也要做运维相关的工作。因为没有额外的运维团队负责监控，所以开发人员需要兼顾起对微服务的随时响应。

技术蔓延

第二个挑战是技术蔓延，这个与组织结构的调整有一定关系。康威定律和反康威定律预言了组织和微服务的扩张，不过在微服务架构里还有第二种蔓延（跟技术、工具相关）也是不可避免的。技术的蔓延会以多种形式出现，我们将会提到几种常见的形式。

当我们面对一个包含上千个微服务的大型微服务生态系统时，就很容易理解为什么微服务会导致技术蔓延。假设每个微服务由一个包含 6 个开发人员的开发团队来负责，每个开发人员使用自己的开发工具、软件包和编程语言，每个开发团队有自己的部署方式，有自己的监控度量指标，有自己的内部和外部依赖项，有自己的运行脚本，等等。

如果你拥有上千个这样的团队，这意味着在一个系统内，同一件事情有上千种做法，有
23 上千种部署方式，有上千个软件包需要维护，有上千种监控方式，有上千种测试方法和上千种处理失效的办法。阻止技术蔓延的唯一方法是对微服务生态系统的每一个层进行标准化。

另一种技术蔓延与开发语言有关。微服务给开发者带来太多的自由，开发者可以随意选择他们喜欢的开发语言和开发库，这也是微服务最令人诟病的一点。虽然不管从原则方面还是从现实来方面来看，这样都是没有问题的，但随着微服务生态系统的发展，情况会变得越来越糟糕。试想一下这样的场景，我们有一个包含了两百个微服务的生态系统，它们有些是用 Python 写的，有些是用 JavaScript 写的，有些是用 Haskell 写的，有些是用 Go 写的，还有一些是用 Ruby、Java 和 C++ 写的。对于这个生态系统里的每一个工具、系统和服务，我们都需要为每一种语言开发一个版本。

仔细想想，为了能够支持每一种开发语言，需要做大量的维护和开发工作，很少有公司能够承担得起这么大的资源开销。选择一小部分开发语言，并确保所有的开发库和工具都能够支持它们，这个比试图支持大量的开发语言要来得实际。

最后一种技术蔓延是技术债务。技术债务是因为前期急于求成，没有以正确或最优的方式完成某些工作，导致后期需要重做这些工作或者修复之前的工作。在这种模式下，微服务开发团队可以很快地开发出很多新的功能特性，而技术债务也会在背后悄悄地堆积起来。当出现问题时，在对问题进行调查之后，开发团队一般会给出并非最好的解决方案：微服务开发团队关心的是当下能否快速地解决问题，他们寄希望于未来能够出现更好的解决方案。

更多失效的可能性

大型分布式微服务系统包含了大量持续变化的小型服务。面对这种复杂的系统，我们需要接受系统里的组件必然会失效的事实，而且它们会经常性地失效，没有人能够对此做

出预测。这就是我们要面对的第三个挑战：微服务架构给系统带来了更多失效的可能性。

为此，我们需要为失效做好准备，当发生失效时我们要想办法减少损失，而且需要对每个组件和整个生态系统的极限和边界进行测试，这些内容将在第 5 章中进行介绍。不过，你一定要明白，不管你运行过多少弹性测试，不管你排除过多少故障，你仍然无法逃脱系统会继续失效的命运——你唯一能做的就是为这些失效做好准备。

资源竞争

跟自然界的生态系统一样，在微服务生态系统里也存在激烈的资源竞争。每个工程组织所拥有的资源都是有限的。他们的工程资源有限（团队、开发人员），硬件和基础设施有限（物理主机、云硬件、数据库存储等），而且每一个资源对公司来说都很昂贵。

如果你的微服务生态系统有大量的微服务和一个大型的应用平台，那么这些团队争夺硬件和基础设施资源是不可避免的：每个服务、每个工具都将变得很重要，而且大量的资源需求迫在眉睫。

类似的，当应用平台团队向微服务团队收集需求时，微服务团队总是说他们的需求是最重要的，而且一旦这些需求没有被接受，他们就会感到失望（有可能还会沮丧）。这种工程资源上的竞争甚至会让团队之间产生仇恨。

最后一种资源竞争可能是最显而易见的：经理之间的竞争、团队之间的竞争以及工程部门和组织之间关于工程人员的竞争。虽然计算机相关专业的毕业生在不断增加，开发阵营也在不断壮大，但好的开发人员仍然不好找，他们成了稀缺资源。当成百上千个团队在竞争一两个名额的时候，每个团队都说自己比其他团队更需要这个名额。

虽然资源竞争无法避免，但还是有一些办法可以减少竞争。最有效的办法大概就是把团队按照它们的重要程度来分类，然后按照团队的优先级和重要程度来分配资源。不过这样做也有不足之处，因为这会导致某些工具开发团队人员配备落后，那些重度依赖未来技术（比如新的基础设施技术）的项目会被抛弃。

第 2 章 <25>

生产就绪

微服务架构给开发人员提供了很大的自由空间，不过要保证整个微服务生态系统的可用性，生态系统里的每一个微服务都要达到很高的架构标准、运维标准和组织标准。这一章将探讨微服务标准化所面临的挑战，解释什么是微服务的可用性，描述微服务的 8 项生产就绪标准，并提供一些如何在工程组织里实现生产就绪标准的建议。

微服务标准化的挑战

单体应用的架构在它的软件生命周期之初就已成型。对于很多应用程序来说，它们的架构在公司建立之初就已经被设计好了。随着公司业务的发展，应用程序的规模也在扩大，开发人员时常感觉到那些在应用设计之初所做的决定对他们形成了束缚。他们无法自由地选择开发语言，无法自由地选择开发库，无法自由地选择开发工具，他们需要做大量的回归测试，确保新增的功能特性不会破坏应用程序的完整性。任何针对单体应用的重构都会受到原始架构的约束，原始的架构决定着应用程序的未来。

微服务架构为开发人员提供了很大的自由空间。他们不再局限于过去的架构，他们可以按照自己的想法设计服务，还可以自由地选择开发语言、数据库、开发工具，等等。微服务开发人员对微服务架构的基本概念烂熟于心：开发一个应用，让它只做一件事情，<26> 并且把它做到最好。按照自己的想法，做任何该做的事情，只要确保可以把它做好。

从理论上说，这种关于微服务开发的理想化想法是没有问题的，不过不是所有的微服务都是一样的。每个微服务都是微服务生态系统的组成部分，而且它们之间存在复杂的依赖关系。如果你有 100 个，或者 1000 个，甚至 10000 个微服务，它们当中的每个个体在整个生态系统里扮演的都只是一个很小的角色。微服务之间必须无缝地进行交互，更重要的是，任何一个微服务都不应该破坏整个生态系统的完整性。如果一个微服务生态系统要想变得足够好，它必须达到一定的标准，而且它的每个部分也必须达到相同的标准。

对于某个特定的微服务开发团队来说，如果我们知道他们的服务将要扮演怎样的角色，那么对他们提出需求并制订标准会相对容易一些。我们可以说："你们的服务必须提供功能 X、功能 Y 和功能 Z，为了能够提供这些功能，你们必须满足要求 S"，然后向这个团队提出需求。不过，这种方式很显然是不能被大规模使用的，而且我们忽略了一个很重要的事实：每个微服务只不过是大型分布式生态系统的一个组成部分。我们必须为所有的微服务定义标准，它们必须能够被应用在每一个微服务上，使得它们可以被量化，并产出可度量的结果。于是生产就绪的概念应运而生。

可用性：标准化的目标

在微服务生态系统里，可用性的服务等级协议（SLA）是最常见的一个可以衡量服务成功与否的标准：如果一个服务具有高可用性（也就是说只有很少的宕机时间），那么我们可以很自信地（有一点点夸张）说这个服务运行良好。

衡量可用性很简单，只需要考虑三个指标：uptime（微服务正常工作的时间）、downtime（微服务无法正常工作的时间）和微服务总运行时间（uptime+downtime）。用 uptime 除以总运行时间（uptime+downtime），就可以得到可用性指标。

可用性并不是微服务标准化原则，而是目标。它之所以不能作为标准化原则，是因为它并没有提供如何架构、构建或运行微服务的指南，因为让开发人员把微服务做到可用却不告诉他们怎么做（或者为什么要这么做）是毫无道理的。可用性不包含任何具体可操作的步骤，不过在接下来的章节我们将看到，需要经过一些步骤才能构建出一个符合可用性标准的微服务。

如何计算可用性

可用性通过"9"的个数来衡量，也就是服务可用时间的百分比。例如，一个 99% 可用的服务就具备了"2 个 9 的可用性"。

这种方法很管用，因为它会告诉我们服务允许的宕机时间。如果你的服务要求 4 个 9 的可用性，也就是允许服务每年有 52.56 分钟不可用，每个月 4.38 分钟不可用，每周 1.01 分钟不可用，每天 8.66 秒不可用。

下面是可用性从 99% 到 99.999% 所对应的宕机时间列表。

99% 可用性：（2 个 9）

- 3.65 天 / 年

- 7.20 小时 / 月
- 1.68 小时 / 周
- 14.4 分钟 / 天

99.9% 可用性：(3 个 9)

- 8.76 小时 / 年
- 43.8 分钟 / 月
- 10.1 分钟 / 周
- 1.44 分钟 / 天

99.99% 可用性：(4 个 9)

- 52.56 分钟 / 年
- 4.38 分钟 / 月
- 1.01 分钟 / 周
- 8.66 秒 / 天

99.999% 可用性：(5 个 9)

- 5.26 分钟 / 年
- 25.9 秒 / 月
- 6.05 秒 / 周
- 864.3 毫秒 / 天

生产就绪标准

生产就绪的基本概念是说，如果一个应用程序或服务被证实可以处理生产环境的业务流量，那么它就被认为是生产就绪的。当我们说一个应用程序或微服务是“生产就绪”的时候，表示我们对它充满了信任：我们相信它的行为是正常合理的，相信它的表现是可靠的，相信它会在极少宕机的情况下处理好任务。生产就绪是微服务标准的主要原则，也是达成微服务生态系统可用性的关键因素。

不过，只是知道生产就绪的概念是远远不够的，概念本身对我们来说不会产生多大作用。我们必须确切地知道一个服务要达到生产就绪标准并具备处理生产环境业务流量能力需要满足哪些条件。这种信任不是能够随意获得的，必须努力去达成。每个微服务、应用程序和分布式系统必须遵循这些原则，不以原则为先导的标准是没有意义的。

总的来说，微服务需要遵循 8 个原则。每个原则都是可以被量化的，每个原则都会提供

一系列可操作的步骤，并产出可衡量的结果。它们分别是：稳定性、可靠性、伸缩性、容错能力、灾备能力、高性能、监控能力和文档化。这些原则组合在一起，共同造就了微服务的可用性。

从某种程度上说，可用性是生产就绪微服务的一个自然属性。一个可伸缩的、可靠的、可容错的、高性能的、可监控的、文档化的和具有良好灾备的微服务就具备了可用性。这 8 个原则中的任何一个都无法单独保证可用性，但把它们放在一起作为构建微服务的架构和运维指南，就可以保证构建出来的系统在生产环境具有高可用性。

29

稳定性

微服务架构为开发人员带来了开发上的自由，并加快了部署的速度。每天都可以开发部署新的功能，缺陷很快被修复，旧技术被新技术所代替，过时的微服务被重写，旧版本的微服务被下架。快速的变更会导致稳定性的下降，有缺陷的代码或其他严重的错误是微服务生态系统出现问题的主要根源。

处理好微服务的变更，提高稳定性，是达成可用性的关键步骤。对于一个稳定的微服务来说，任何开发、部署、技术更新、服务下架都不应该降低微服务本身和整个生态系统的稳定性。我们可以为每个微服务定义稳定性标准，以便减少微服务变更给稳定性带来的副作用。

为了减少可能在开发过程中出现的问题，可以采用稳定的开发流程。为了降低因开发带来的不稳定性，我们要小心翼翼地部署微服务，从 staging 环境到 canary 环境（使用生产环境 2%~5% 的服务器），再到生产环境。为了避免因采用新技术和下架旧服务对其他服务的可用性造成影响，我们要严格遵循稳定的更新换代流程。

稳定性需求

构建稳定的微服务要求：

- 稳定的开发周期。
- 稳定的开发流程。
- 稳定的更新换代流程。

稳定性需求的细节将在第 3 章进行讨论。

可靠性

稳定性无法单独保证微服务的可用性，一个可用的服务也必须是可靠的。一个微服务需

要得到客户端、依赖项和整个微服务生态系统的信任才能算得上可靠。一个可靠的微服务必须已经获得了信任，这是它走向生产环境的必要条件。 <30>

稳定性的目的在于减少微服务发生变更所带来的副作用，而可靠性与信任有关。稳定性和可靠性有着不可分割的密切联系。稳定性总是伴随着可靠性，例如，开发人员不仅要确保部署流程的稳定，还要保证每一次部署对其他客户端和依赖项来说都是可靠的。

与稳定性需求一样，可靠性也可以被分为若干个需求。例如，要确保集成测试覆盖全面，staging 和 canary 能够部署成功，我们可以相信每一个引入到生产环境的变更不会包含任何可能破坏客户端和依赖项的错误，从而确保部署流程是可靠的。

可以通过提升微服务的可靠性来保证可用性。我们可以对数据进行缓存，其他服务就可以很快地读取到这些数据。服务的高可用可以帮助其他服务保证它们的 SLA。而为了保证自身的 SLA，避免因依赖项的可用性问题给我们造成不利影响，我们需要实现防御性缓存。

可靠性的最后一个要求与路由和服务发现有关。可用性要求不同服务之间具备可靠的通信和路由：要求有准确的心跳检测，要求请求和响应必须能够到达它们的目的地，要求错误必须被恰当地处理。

可靠性需求

构建可靠的微服务要求：

- 可靠的部署流程。
- 对依赖项可能出现的故障准备应对措施。
- 可靠的路由和服务发现。

可靠性需求的细节将在第 3 章进行讨论。

伸缩性

微服务的业务流量模式很少是固定不变的，成功的微服务（或微服务生态系统）有一个显著的特点，它们的业务流量总是呈稳健的增长势头。在构建微服务时需要为此做好准备，让它们能够轻松地应对流量的增长，并且可以灵活地进行伸缩。微服务如果无法随着流量伸缩，它将导致延迟的增加和糟糕的可用性，在极端情况下还会频繁地出现故障和宕机。伸缩性是可用性的基础，也是第三个生产就绪标准。 <31>

一个可伸缩的微服务可以同时处理大量的任务和请求。要确保微服务的伸缩性，我们需要知道它在质（比如它是否会在页面访问或客户订单方面进行伸缩）和量（比如它每秒

可以处理多少个请求）两个方面的增长规模。在了解了它的增长规模之后，我们就可以对未来的容量进行规划，并识别出资源瓶颈和资源需求。

除此之外，微服务处理业务流量的方式也应该是可伸缩的。微服务需要随时做好处理突发流量的准备，并小心地处理这些流量，防止它们拖垮整个服务。说起来容易做起来难，不过事实就是如此，如果流量的处理方式无法进行伸缩，开发人员将会眼睁睁地看着他们的微服务生态系统走向崩塌。

生态系统的其他部分会带来额外的复杂性，所以也要为来自系统其他部分的流量增长做好准备。同样的，微服务在增长自身流量时也要向它的依赖项发出通知。跨团队的合作沟通是伸缩性的基础，经常性地与上游和下游的微服务团队就服务的伸缩性需求、状态和瓶颈方面的问题进行沟通，可以确保每个相互依赖的服务为应对流量增长和潜在风险做好了准备。

微服务存储和处理数据的方式也需要具备伸缩性，这点也很重要。一个可伸缩的存储方案与微服务的可用性息息相关，它是生产就绪系统最重要的组件。

伸缩性需求

构建可伸缩的微服务要求：

- 定义好质和量两方面的增长规模。
- 定位资源瓶颈和需求。
- 准确的容量规划。
- 可伸缩的流量处理能力。
- 依赖项的伸缩。
- 可伸缩的数据存储。

伸缩性需求的细节将在第 4 章进行讨论。

32 容错和灾备

即使一个最普通的微服务生态系统也是相当复杂的。我们都知道，复杂的系统容易出现问题，而且会经常出现，在微服务生命周期的任何时间点都有可能出现问题。微服务不是独立存在的，它们存在于依赖关系链里，这些关系链是大型微服务生态系统的组成部分。复杂度会随着微服务的数量增加而呈线性地增长，要保证整个生态系统的可用性，我们需要制订更多的生产就绪标准，这就要求生态系统里的每个微服务必须具备容错能力和灾备能力。

一个具备容错和灾备能力的微服务可以承受来自内部和外部的故障。内部故障是微服务自身造成的，例如，没有经过严格测试的代码缺陷导致的部署失败，它们会影响整个生态系统。而外部故障，比如数据中心宕机或糟糕的管理配置，它们会影响每个微服务和整个系统的可用性。

我们可以为这些潜在的灾难性故障适当地（不一定是全面地）做好应对准备。定位故障场景是构建一个具备容错能力的微服务的首要条件。在定位出这些故障场景之后，就要开始制订应对策略。我们需要为微服务生态系统的每一个层制订策略，而且任何可重用的策略都需要在整个组织范围内进行讨论，并把它们标准化。

在组织层面对故障的处理进行标准化，这意味着每个微服务、基础设施组件和整个微服务生态系统的故障都需要被纳入到一些程序里，这些程序必须被小心地执行，并且易于理解。我们要以一种谐调的、有计划的和深入沟通的方式来处理故障事件。除此之外，如果故障事件响应过程经过良好的定义，那么组织就可以避免长时间的故障，并保持服务的可用性。如果每个开发人员清楚地知道在出现故障时，他们该做些什么，知道该怎么快速地解决问题，知道如何对超出他们能力范围的问题进行升级，那么处理故障的时间就会缩短很多。

对故障进行预测意味着要比故障的出现先行一步，并为之做好应对准备。我们需要对微服务、基础设施和整个生态系统做各种可用性测试。这个可以通过多种弹性测试来实现。第一步是代码测试（包括单元测试、回归测试和集成测试）。第二步是负载测试，这个主要测试微服务和基础设施组件在极端的流量压力下是否仍然保持可用。最后一步是最重要的混沌测试，这个测试在生产环境验证（安排好的或随机的）各种故障场景，确保微服务和基础设施组件能够达到生产就绪的标准。

33

容错和灾备需求

构建具备容错能力和灾备能力的微服务要求：

- 识别潜在的故障场景，并做好应对准备。
- 识别并解决单点故障问题。
- 应用故障探测和补救策略。
- 通过代码测试、负载测试和混沌测试验证系统的弹性。
- 管理好业务流量。
- 快速处理故障。

容错和灾备需求的细节将在第 5 章进行讨论。

高性能

在微服务生态系统来里，伸缩性（之前提到过）与微服务能够处理的请求数量有关。而高性能是另一个生产就绪标准，它指的是微服务对请求的处理速度。一个高性能的微服务可以使用适当的资源（硬件和基础设施组件）快速地处理请求，高效地执行任务。

需要大量网络调用的微服务不能算是高性能的。如果有些场景可以使用异步（非阻塞）任务处理代替原先的同步任务处理，以便提升性能和服务的可用性，那么原先的同步服务就不能算是高性能的。识别并解决这些性能问题是生产就绪标准的一个严格要求。

类似的，为微服务分配过量的资源（比如CPU）不仅不会为效率的提升带来任何好处，反而会降低性能。如果这个问题不能在微服务层面得到解决，它会在生态系统层面给我们带来麻烦。不充分的硬件资源利用影响的是系统的硬件层。在资源的合理计划和过度

分配之间有一道分界线，我们要把这两者结合起来综合考虑，避免对微服务的可用性造成破坏，并且让资源的使用更加合理。

高性能需求

构建高性能的微服务要求：

- 恰当的可用性SLA。
- 恰当的任务处理方式。
- 对资源的合理利用。

高性能需求的细节将在第4章进行讨论。

监控

要保证微服务的可用性，还需要适当的监控。做好监控需要三个组件：关键信息的日志、能够准确反映服务健康状况且易于理解的用户图形展示界面（仪表盘）以及有效且具有可操作性的关键性指标告警。

每个微服务从一开始就需要记录日志。哪些信息需要被记录因每个微服务而异，不过记录日志的目的是一样的：在出现问题时——有可能是之前的部署造成的缺陷——需要从这些日志里找到出错的原因。在微服务生态系统里，不建议使用微服务版本管理，所以无法准确地知道缺陷所对应的代码版本。代码经常被改动，每周被部署好几次，新功能不断在增加，依赖关系也一直在变化，但日志却保持不变，它们始终保留着可以用来定位问题的信息，所以要确保日志里包含了这些信息。

所有关键性度量指标（包括硬件使用情况、数据库连接、响应时间和平均响应时间以及API端点的状态）应该被实时地以图形化的方式展示在仪表盘上。要监控好生产就绪的微服务，仪表盘是一个很重要的组件：仪表盘可以让人一眼就看出微服务的健康状况，开发人员可以从仪表盘上看出一些反常的情况，比如那些还达不到触发告警条件但却十分诡异的情况。如果使用得当，开发人员从仪表盘上一眼就可以看出微服务是否工作正常，不过开发人员不应该使用仪表盘作为故障探测的主要手段，故障探测和回滚的过程应该是完全自动化的。 35

真正的故障探测是通过告警来完成的。所有的关键性度量指标都必须被纳入告警范围，至少应包括如下指标：CPU和RAM的使用情况、文件描述符的数量、数据库连接的数量、服务的SLA、请求和响应消息、API端点的状态、错误和异常、服务依赖项的健康状况、数据库的其他信息、正在处理的任务数量（如果有的话）。

一般来说，我们需要为每一个度量指标单独设置相应的阈值和告警线，一旦出现偏离基准（达到了告警线或阈值）的情况，正在待命的开发人员就会收到告警。阈值的设定要恰当：高到可以避免因频繁的告警造成干扰，但同时要确保可以捕捉到问题。

告警需要具备可操作性。如果告警不具备可操作性，它就没有意义，它会浪费大家的时间。针对每一个具备可操作性的告警，应该有相应的操作手册。例如，如果一个告警提醒某种异常数量超标，那么一个待命的开发人员可以根据操作手册上的说明来解决问题。

监控需求

构建可监控的微服务要求：

- 适当的日志和堆栈跟踪信息。
- 设计良好且易于理解的仪表盘，可以准确地反映服务的健康状况。
- 有效且具有可操作性的告警操作手册。
- 开发人员轮班待命。

监控需求的细节将在第6章进行讨论。

文档化

微服务架构可能会带来更多的技术债务，这也是采用微服务架构所要面临的一个关键性挑战。一般来说，技术债务与开发速度有关系：服务迭代、变更、部署得越快，就会有越多的临时解决方案和补丁出现。在组织层面对微服务进行文档化，可以减少技术债务，同时也会减少困惑、增长知识、加强对整体架构的理解。 36

减少技术债务并不是让文档化成为生产就绪标准的唯一原因，如果只是这样，那么文档化也只能算得上是一个马后炮（虽然很重要，但仍然只是一个马后炮）。实际上，与其他的生产就绪标准一样，文档化和对文档的理解直接影响着微服务的可用性。

为了说明这一点，我们可以想象一下开发人员在一起工作的情景，他们彼此分享着对微服务的理解。你可以试着让你的团队坐在一个房间里，在一块白板前面，让他们把系统架构和微服务相关的重要细节画在白板上。我相信你会对结果感到吃惊，你会发现他们对微服务的理解不在一个频道上。有人知道其他人所不知道的，也有人对微服务的理解让你不禁要怀疑他是否在开发这个微服务。在审查代码变更、更换新技术、增加新功能的时候，对各方面知识理解的不一致会导致微服务无法达到生产就绪标准，严重的瑕疵会破坏微服务的可靠性。

这个问题是可以避免的，只要让每一个微服务遵循严格的文档标准即可。文档里需要包含微服务所有有用的信息，比如架构图、开发上手指南、请求消息流和 API 端点的相关细节以及问题处理手册。

我们可以通过多种途径来了解一个微服务。第一种可以像之前所说的那样：跟开发团队一起坐在一个房间里，让他们把微服务的架构图画出来。因为开发速度一直在加快，微服务在它们整个生命周期的不同时刻都发生着巨大的变化。把架构评审作为开发流程的一部分，这样可以保证整个团队了解微服务的每个变更。

第二个了解微服务的途径是，我们需要了解生产就绪标准本身。在很大程度上，对微服务的了解在于它是否符合生产就绪标准，或者已经达到生产就绪标准的哪一等级。我们可以通过很多种方式来做出判断，其中的一种就是对微服务进行评审，看它是否满足生产就绪标准，如果不是，那么就创建一个可以把它带向生产就绪状态的线路图。评审过程可以在整个组织里自动进行。在下一章我们将介绍如何在采用了微服务架构的组织里实现生产就绪标准，我们将深入探讨这方面的内容。

文档化需求

构建文档化的微服务要求：

- 详细的、最新的、集中式的文档，它包含了微服务所有的相关信息。
- 在开发人员、团队、整个生态系统层面了解微服务。

文档化需求的细节将在第 7 章进行讨论。

实现生产就绪标准

我们现在有了一组可以应用在每个微服务上的标准，每个标准都有自己的特定要求。任何满足了这些要求的微服务都可以在生产环境处理业务流量，而且可以保证高度的可用性。

标准是有了，那么我们如何在真实的微服务生态系统里去实现它们呢？从理论到实践总是有一定难度的。不过，生产就绪标准的影响力和它们的特定要求取决于它们的使用场景和粒度：这两者都可以被用于任何一个生态系统，不过还无法细化到可以提供实现的具体策略。

标准化要求在组织的各个层面展开，不仅要从上到下，还要求从下到上。在领导（管理和技术方面）层面，这些原则需要被作为整个工程组织的架构要求来执行。而在员工层面，对于每个开发团队来说，他们需要接受并执行这些标准。最关键的是，标准化不应该被视作一堵限制开发和部署的墙，而是要把开发和部署带向生产环境的指南。

可能很多开发人员会抗拒标准化。他们会争辩说，采用微服务架构不就是为了速度、自由和效率吗（为什么还要制订这些标准）？确实，微服务架构为开发团队带来了自由和效率，但这也是生产就绪标准发挥作用的地方。当一个故障让服务失效，当一个糟糕的部署破坏了微服务的可用性，当一个本该可以通过适当弹性测试来避免的故障拖垮了整个微服务生态系统，那么开发上的自由和效率也就慢慢失去了意义。如果我们对过去五十年的软件开发有所了解，就会知道，标准化会带来自由并减少混乱。就如 Brooks 在他的《人月神话》里所说的那样——“形式即自由”。

在整个工程组织接纳了生产就绪标准之后，下一步是详细制订每个标准的具体要求。本书中所提到的要求都是很笼统的，我们需要结合具体的上下文环境和特定的组织细节以及实现策略来理解它们。我们要做的工作是针对每个生产就绪标准和它的要求提出实现方案。例如，如果组织的微服务生态系统里有一个自助部署工具，那么我们就需要实现一个稳定可靠的部署流程，这个流程需要考虑工具的工作原理。重新构建内部工具或给它们添加新的功能也需要遵循这样的方式。

开发人员、团队主管、管理层或运营（系统、DevOps 或网站可靠性方面的运营）人员都可以实现这些要求，他们可以自己判断一个微服务是否满足这些要求。据我所知，Uber 和其他一些公司已经采用了生产就绪标准，这些标准的实现和执行由网站可靠性部门来负责。网站可靠性部门一般负责服务的可用性，所以在整个生态系统内推广这些标准刚好和他们的职责不谋而合。当然，这并不是说开发团队在这方面就没有责任了。网站可靠性部门负责在生态系统内驱动标准的实施，但开发人员也需要负起相应的责任。

构建和维护一个生产就绪的微服务生态系统不是一件容易的事情，不过如果做得好，我们可以从中得到很大的好处，微服务的可用性会因此得到保证。生产就绪标准为我们提供了可衡量的结果，开发团队可以看到他们所依赖的服务是可信任的，并具备了稳定性、可靠性、容错能力、灾备能力、高性能、监控能力和文档化。

第 3 章 41

稳定性和可靠性

生产就绪的微服务同时具备了稳定性和可靠性。微服务个体和整个微服务生态系统都在不断演化，任何尝试改进微服务稳定性和可靠性的努力都在把整个生态系统的可用性推向新的高度。在这一章中，我们将探讨几种方式用于构建稳定可靠的微服务，包括标准化开发流程、构建部署管道、理清服务依赖关系并做好依赖失效保护、构建稳定可靠的路由和服务发现机制，以及建立过期微服务和过期 API 端点的弃用规程。

微服务稳定性和可靠性的原则

微服务架构加快了开发速度。微服务带来的自由让整个生态系统一直处于持续变化的状态。每天都有新功能添加进来，每天都有好几次构建和部署，新技术以惊人的速度替换旧技术。这种自由和灵活性促进了创新，这种创新是切实可见的，但这是以付出高昂成本为代价的。

如果微服务生态系统里的任何一个部件变得不稳定或不可靠，那么创新进程、开发效率、技术迭代以及持续变化的微服务生态系统都会受到影响。在某些情况下，如果一个构建过程出现错误或其中的一个关键性业务服务组件出现了缺陷，那么后续的部署有可能会导致整个系统无法正常工作。

稳定的微服务不会在开发、部署、更新技术或弃用旧服务时给整个生态系统带来稳定性方面的影响。我们需要采取一些措施来应对这些变化可能带来的负面影响。可靠的微服务可以被其他服务和整个生态系统所信赖。稳定性和可靠性齐头并进，因为每一个稳定性的要求都需要以可靠性作为前提（反之亦然）。例如，对于微服务的客户端或依赖项来说，在一个稳定的部署过程里，任何新的部署都不能破坏服务的可靠性。 42

我们可以采取一些措施来保证微服务的稳定性和可靠性。我们可以把开发周期标准化，

设计合理的部署流程，代码的变更在推向生产环境之前需要强行经过几道门坎儿。我们要管理好依赖关系，为依赖失效做好准备。路由和服务发现通道里需要包含心跳检测机制、恰当的路由机制和回路断路机制，以便应对各种异常的业务流量场景。最后，微服务及其端点的弃用不应该对其他微服务造成任何影响。

一个生产就绪的微服务是稳定且可靠的

- 它有一个标准化的开发周期。
- 它的代码需要经过初步检查、单元测试、集成测试以及端到端的测试。
- 它的测试、打包、构建和发布流程是自动化的。
- 它有标准化的部署管道，包括 staging 阶段、canary 阶段和生产阶段。
- 它的客户端是已知的。
- 它的依赖项是已知的，而且是有备份的，还有可选的回退方案以及缓存，以防出现依赖项失效。
- 它有稳定可靠的路由和服务发现机制。

开发周期

开发人员是第一批需要对微服务的稳定性和可靠性负起责任的人。大多数微服务故障是由代码的缺陷引起的，这些缺陷在开发阶段、测试阶段或部署阶段没有被检查出来。解
43 决这类故障一般是回退到上一个稳定版本，把代码恢复原状，然后重新部署。

为不稳定不可靠的开发付出的代价

微服务生态系统不是“狂野的西部”。每一个故障、每一个突发事件以及每一个缺陷都会让公司耗费数千（最好不是百万）美元的成本，这些成本体现在工程时间和利润损失上。开发周期（在部署管道里也一样）里应该包含防护措施，在进入生产环境之前把缺陷扼杀在襁褓之中。

稳定可靠的开发周期包含了几个步骤，如图 3-1 所示。

首先，开发人员对代码做出变更。我们一般会从代码中心仓库（git 或 svn）拉出一个分
44 支，开发人员在分支上进行修改，然后运行单元测试和集成测试。这个步骤可以在任何地方进行，例如在开发人员的开发机上或者开发环境的服务器上。一个可靠的开发环境相当于生产环境的镜像，在对微服务进行测试时，如果需要调用其他服务或者访问数据库，那么这个环境就变得尤为重要。

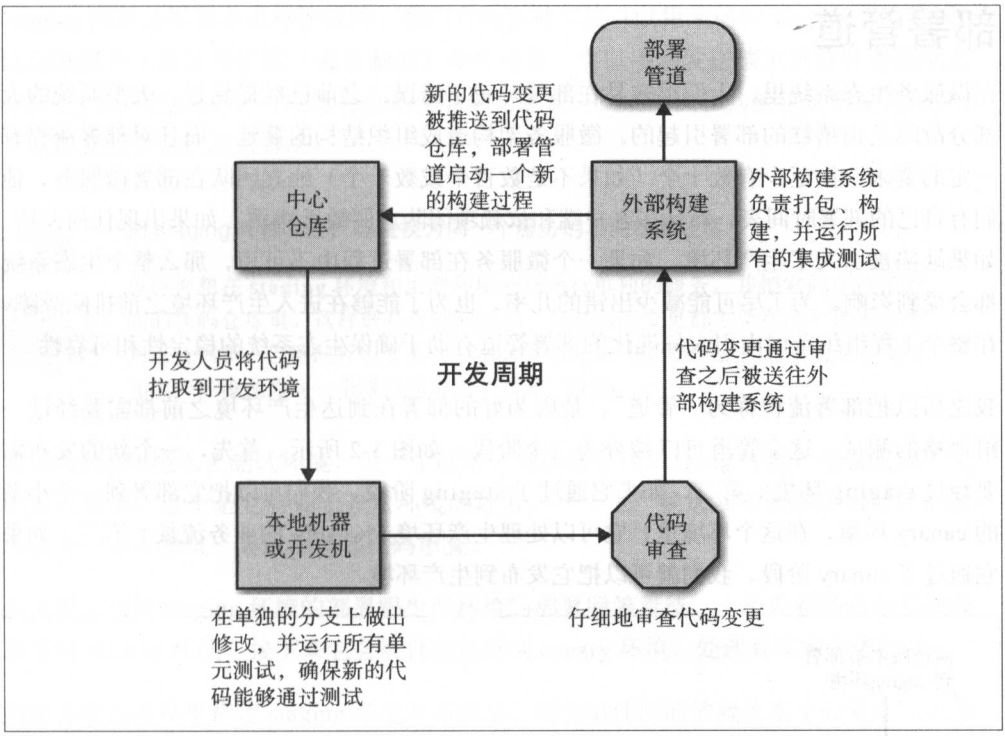


图3-1：开发周期

在代码提交到中心仓库之后，其他人需要对代码进行审查。如果代码通过了审查，并且通过了初步检查、单元测试和集成测试，那么就可以被合并到代码仓库里（第 5 章会详细介绍代码的初步检查、单元测试和集成测试）。只有在完成了上述所有步骤之后，代码才能被引入部署管道。

代码审查前的测试

在进行代码审查之前对代码进行初步检查、单元测试、集成测试和端到端测试可以确保缺陷不会被带入生产环境。开发人员在分支上进行开发，在提交代码之后，需要运行所有的测试，通过测试之后才能到达代码审查阶段（或者到达构建阶段）。

在第 1 章介绍微服务生态系统第 4 层的内容时，我们就已经提到，在开发周期和部署管道之间会发生很多类似的过程。在进入部署管道之前，新的版本发布需要被打包、构建和测试。

部署管道

在微服务生态系统里，人们很容易在部署时犯下错误。之前已经提到过，大型系统的大部分故障是由糟糕的部署引起的。微服务架构造成组织结构的蔓延，而且对部署流程有一定的要求：你至少有数十个（如果不是数百个或数千个）独立团队在部署微服务，他们有自己的部署时间表，而且在客户端和依赖项团队之间缺乏协调。如果出现任何差错，如果缺陷被引入了生产环境，如果一个微服务在部署过程中不可用，那么整个生态系统都会受到影响。为了尽可能减少出错的几率，也为了能够在进入生产环境之前排除故障，在整个工程组织范围内引入标准化的部署管道有助于确保生态系统的稳定性和可靠性。

我之所以把部署流程称为“管道”，是因为好的部署在到达生产环境之前都需要经过一组严格的测试。这个管道可以被分为三个阶段，如图 3-2 所示。首先，一个新的发布需要经过 staging 环境；第二，如果它通过了 staging 阶段，我们可以把它部署到一个小型的 canary 环境，在这个环境里，它可以处理生产环境 5%~10% 的业务流量；第三，如果它通过了 canary 阶段，我们就可以把它发布到生产环境。

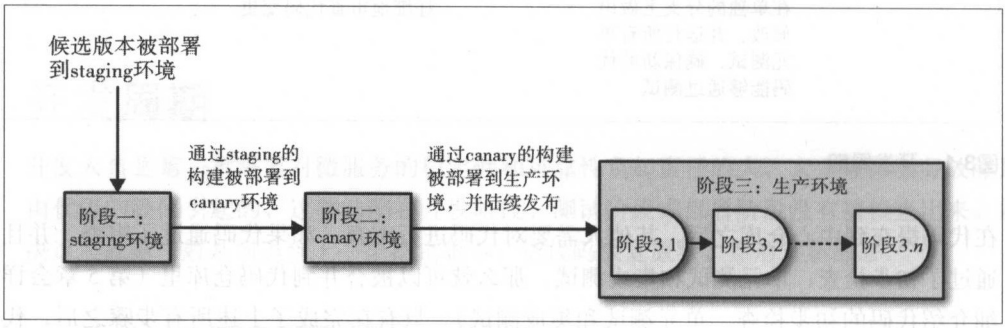


图3-2：稳定可靠的部署管道的几个阶段

staging

每一个版本的发布首先会被部署到 staging 环境。staging 环境应该近似生产环境：它要反映真实生产环境的状态，只不过没有真实的业务流量。staging 环境一般不会达到生产环境那样的规模（比如，它们一般不会有跟生产环境一样多的服务器，这也就是所谓的*主机平价*），因为运行两个同样大规模的生态系统需要很大的硬件开销。一些工程组织认为，要准确地复制一个稳定可靠的生产环境，通过使用主机平价（host parity）来构建一个 staging 环境是唯一可行的途径。

通过计算 staging 环境规模与生产环境规模的百分比来决定应该使用多少硬件，这对于大部分工程组织来说已经足够准确了。staging 环境所需要的处理能力取决于我们是如何

在 staging 阶段对微服务进行测试的。我们有很多种方法可以用来运行 staging 环境测试：可以在微服务上触发模拟的（或录制的）业务流量，可以手动发送请求到微服务的端点并分析它们的响应结果，可以运行自动化的单元测试、集成测试和其他特定的测试用例，还可以任意组合这些测试方法。

把 staging 环境和生产环境视为同一个服务的不同部署过程

46

也许你想在 staging 环境和生产环境分别运行单独的服务，并把它们存放在不同的代码仓库里。这样做是没有问题的，只是每次变更都需要在两个代码仓库间进行同步，包括配置的变更（经常会被忘记）。更好的做法是把 staging 环境和生产环境视为同一个服务的不同“部署”阶段。

尽管 staging 环境也是测试环境，但一个版本一旦被部署到 staging 环境，它就会成为生产的候选版本，这个是它有别于开发环境的地方。一个生产候选发布版本必须已经通过初步检查、单元测试、集成测试和代码审查。

开发人员应该把 staging 环境的部署跟生产环境的部署同等看待。一个发布版本如果被成功部署到 staging 环境，接下来它会被自动部署到 canary 环境，处理真实的业务流量。

在微服务生态系统里搭建 staging 环境并不容易，因为组件间的依赖关系十分复杂。如果你的微服务依赖其他服务，那么在它发出请求之后就需要等待其他服务的响应，其他服务需要从数据库读取数据或向数据库写入数据。因为这些复杂性，staging 标准化的结果决定了 staging 环境的好坏。

full staging

部署管道的 staging 阶段可以有多种不同的配置。第一种是 full staging（参见图 3-3），它会运行一个跟生产系统一样的 staging 系统（不一定要用主机平价）。full staging 运行在跟生产环境一样的基础设施上，不过有几个关键的区别，full staging 会使用特定的端口来暴露服务。在一个 full staging 生态系统里，staging 环境只会跟 staging 环境里的服务打交道，它们不会给任何运行在生产环境的服务发送请求，也不会从那里接收任何响应（也就是说，从 staging 环境到生产环境的通信端口被封锁了）。

full staging 要求 staging 环境为每一个微服务提供完善的功能，在新版本发布的时候，微服务之间仍然可以进行交互。staging 生态系统里的微服务交互可以通过运行特定的测试用例来实现，或者就像之前提到的那样，通过触发事先录制好的生产业务流量或模拟的业务流量来进行测试。

47

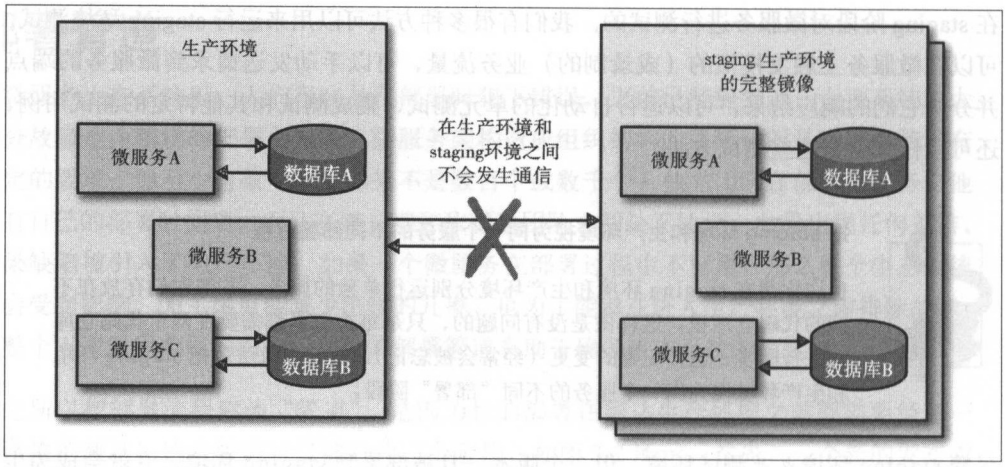


图3-3: full staging

处理 full staging 的测试数据要十分小心：staging 环境不应该拥有对生产环境数据库的写权限，也不建议把生产环境数据库的读权限授权给 staging 环境。因为 full staging 被视为生产环境的一个完整镜像，所以每个 staging 环境应该有自己的数据库。

full staging 的风险

在使用 full staging 环境时要提高警惕，新发布的服务会跟它上下游的服务保持交互，但这可能不能准确反映现实的真实状况。工程组织需要让团队协调好 staging 部署，避免一个服务的部署破坏了其他相关服务所依赖的 staging 环境。

48 partial staging

第二种 staging 环境是 partial staging。从它的名字上可以看出，partial staging 并非生产环境的完整镜像。每个微服务都有自己的 staging 环境，这个环境有一组服务器，它们使用特定的端口。在服务的新版本发布到 staging 环境时，它们会跟上游的客户端以及运行在生产环境里的下游依赖项进行交互（参见图 3-4）。

partial staging 的部署要覆盖到微服务所有的客户端和依赖项端点，要尽可能准确地模拟真实环境的状态。为了做到这点，需要运行特定的 staging 测试，并且每个新加入的功能都需要经过至少一次额外的 staging 测试。

partial staging 的风险

因为 partial staging 环境的微服务会跟生产环境的微服务发生交互，所以对此要十分谨慎。虽然 partial staging 的请求被限定为只读，但一个糟糕的 staging 部署所导致的大量请求仍然会让生产环境的服务出现过载，从而把它们拖垮。

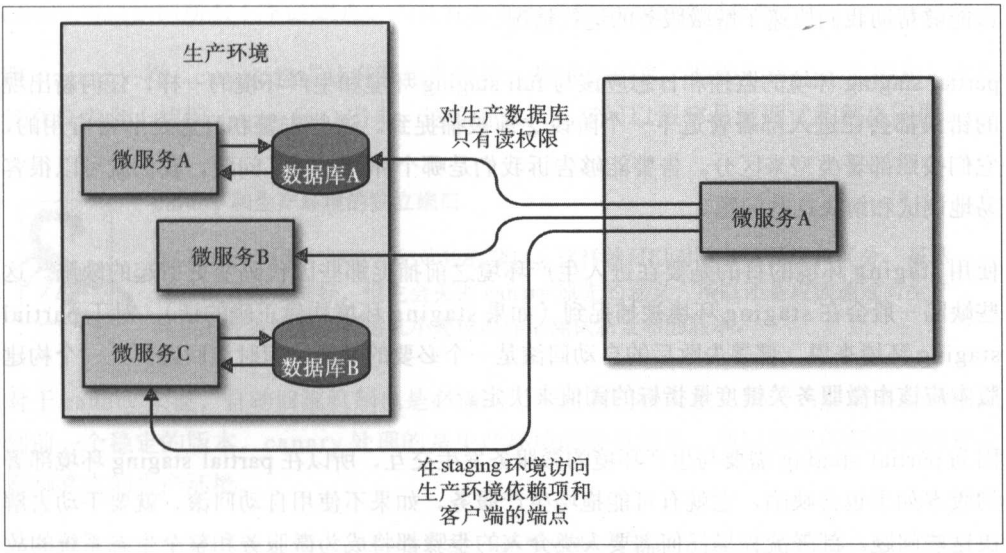


图3-4：partial staging

partial staging 环境对数据库的访问也应该被限定为只读的，也就是说，staging 环境永远不要往生产数据库写入数据。不过，有些微服务可能涉及大量的写操作，但又不可避免地要对这些功能进行测试。对于这种情况，一般的做法是把由 staging 环境写入的数据标记为*测试数据*（也就是*测试租用*）。不过最安全的做法是把数据写入单独的测试数据库，毕竟给 staging 环境开放数据库写权限会对真实的数据造成影响。表 3-1 对 full staging 和 partial staging 环境进行了对比。

49

表3-1：full staging和partial staging环境的对比

	full staging	partial staging
生产环境的完全镜像	是	否
单独的端口	是	是
访问生产服务	否	是
读取生产数据库	否	是
写入生产数据库	否	是
要求自动回滚	否	是

staging 环境（full 或 partial）应该跟生产环境一样，要有仪表盘、监控和日志（第 6 章会介绍监控）。所有的关键度量指标都可以被展示在同一个仪表盘上，不过开发团队可能会为部署流程的不同阶段使用单独的仪表盘：一个 staging 仪表盘，一个 canary 仪表盘，一个生产仪表盘。仪表盘是可配置的，最好把所有阶段部署的度量指标展示在同一个仪表盘上，并按照部署阶段或度量指标来区分。不管最终开发团队决定如何使用仪表盘，构建一个生产就绪的仪表盘这个目标总是不变的：一个生产就绪微服务的仪表盘应

该能够帮助我们快速了解微服务的运行情况。

partial staging 环境的监控和日志应该与 full staging 环境和生产环境的一样，任何新出现的错误都会在进入部署管道下一个阶段之前被捕捉到。这些告警和日志是非常有用的，它们按照部署类型来区分。告警能够告诉我们是哪个环境出现了问题，我们就可以很容易地调试和解决这些问题。

使用 staging 环境的目的是要在进入生产环境之前捕捉那些因代码变更引起的缺陷。这些缺陷一般会在 staging 环境被捕捉到（如果 staging 环境构建正确的话）。对于 partial staging 环境来说，部署失败后的自动回滚是一个必要的功能。何时该回退到上一个构建版本应该由微服务关键度量指标的阈值来决定。

因为 partial staging 需要与生产环境的微服务发生交互，所以在 partial staging 环境部署的版本如果包含缺陷，它就有可能拖垮生产服务。如果不使用自动回滚，就要手动去解决这些问题。部署流程里任何需要人类介入的步骤都将成为微服务和整个生态系统的故障点。

微服务团队在搭建 staging 环境时需要考虑的最后一个问题是：一个新发布版本在进入 canary（然后进入生产环境）之前要在 staging 环境里待多久？这个要取决于 staging 环境的测试情况：如果通过了所有测试，就可以进入下一阶段。

canary

新发布版本在成功部署到 staging 环境并通过所有测试之后就可以进入部署管道的下一个阶段：canary 环境。煤矿工人使用金丝雀作为探测矿井空气质量的工具，这也就是 canary 环境名字的由来：煤矿工人在进入矿井前会带上金丝雀，用来探测空气中一氧化碳的浓度，如果金丝雀死了，说明空气中的有毒成分较高，他们就会离开矿井。把新发布版本部署到 canary 环境里也是基于同样的目的：先部署到生产环境的一小部分服务器上（大概 5% 到 10%），如果没有问题，再部署剩余的服务器。

canary 业务流量分布

如果生产服务部署在多个数据中心、多个区域或者云端，那么 canary 要同时从这些环境里挑选服务器，以便更准确地对生产环境进行取样。

因为 canary 处理的是生产流量，所以它应该被视为生产环境的一部分。它应该要有跟生产环境一样的端口，canary 的主机应该是从生产环境里随机挑选出来的，并能够准确地对生产业务流量进行取样。canary 可以（也应该）有对生产服务的所有访问权限：

canary 可以访问所有上下游端点，而且具有对生产数据库的读写权限。 51

跟 staging 一样，canary 也应该有跟生产环境一样的仪表盘、监控和日志。告警和日志应该在仪表盘上使用 canary 区分出来，这样开发人员就可以很容易地调试和解决问题。

canary 和生产环境的独立端口

为 canary 和生产环境分配单独的端口，这样就可以对业务流量进行区分。虽然这样做看起来挺好，但它会失去 canary 原先的意义，所以还是应该通过对生产环境的业务流量进行随机小规模取样来测试新发布的版本。

对于 canary 来说，自动回滚机制也是必需的：一旦有错误发生，部署系统需要自动回退到前一个稳定的版本。canary 处理的是生产环境的业务流量，所以发生的任何问题都会影响真实的生产环境。

在开发人员认为可以发布到生产环境之前，新发布版本要在 canary 里待多久？答案可以是几分钟、几个小时甚至几天，这完全取决于业务流量。不管你的微服务或业务逻辑有多奇怪，它们的流量总是存在某种模式。在完成一个完整的业务流量周期之前，新发布版本不应该离开 canary 环境。“业务流量周期”需要在组织层面进行标准化定义，但业务流量周期的时间段和具体需求要在微服务层面进行定义。

生产

生产环境处理真实的流量。一个构建版本在经过了开发、staging 和 canary 阶段，接下来就可以部署到生产环境。这是部署管道的最后一步，开发团队应该对他们的构建版本充满信心。到了这个阶段，代码里的任何错误应该都已经被发现和解决了。

每一个即将进入生产环境的构建版本都应该是稳定且可靠的。一个被部署到生产环境的构建版本必须经过详尽的测试，必须经历 staging 和 canary 阶段，而且不能存在任何问题，否则就不能被部署到生产环境。构建版本在通过 canary 阶段之后可以一步到位地部署到生产环境，也可以分阶段循序渐进地部署：开发人员可以按照硬件比例来部署（例如，先部署到 25% 的服务器，然后是 50%，接下来是 75%，最后是 100%）。也可以按照数据中心、区域、国家来部署，或者按照以上几种方式的组合来部署。 52

让稳定可靠的部署成为强制措施

一个生产候选版本在经历了开发阶段、staging 阶段和 canary 阶段之后，造成严重中断的可能性是很小的，因为在它被部署到生产环境之前，大部分缺陷都已经被修复了。这也就是为什么构建一个稳定可靠的微服务需要有一个完整的部署管道。

在有些开发人员看来，部署管道所带来的延迟似乎是没有必要的，因为有了部署管道，他们为修复缺陷或新增功能所写的代码就无法立即部署到生产环境。但实际上，部署管道所带来的延迟是很短暂的，而且可以对其进行灵活调整。不过为了确保可靠性，部署流程还是有必要被作为强制性的标准的。频繁部署微服务会破坏微服务的稳定性和可靠性，也会影响到依赖关系链里的其他微服务——一个每几个小时就发生变更的微服务是不稳定且不可靠的。

如果生产环境出现了严重的缺陷，开发人员会试图跳过部署流程的 staging 和 canary 阶段，直接把修复的代码部署到生产环境。这样做可以很快地解决问题，或许可以为公司减少损失，并避免依赖项发生中断。不过只有在生产环境出现非常严重的缺陷时才允许这么做。如果不进行严格限制，直接部署到生产环境的行为有可能出现泛滥：对于大部分开发人员来说，他们的每个代码变更、每个部署都是很重要的，重要到可以直接跳过 staging 和 canary 阶段，从而破坏了整个微服务生态系统的稳定性和可靠性。如果发生了错误，应该鼓励开发团队把服务回滚到之前的状态，这个状态应该是稳定的，并且没有缺陷。与此同时，开发团队要找出错误发生的原因。

热修复补丁是反模式

有了部署管道，就不应该直接把代码部署到生产环境，除非是紧急情况，不过就算是紧急情况也不建议这么做。跳过部署管道的前面几个阶段很容易在生产环境引入新的缺陷，因为紧急修复代码一般没有经过适当的测试。如果有可能，开发人员应该把服务回滚到上一个稳定的版本，而不是直接往生产环境部署一个紧急修复补丁。

遵循部署管道流程可以获得稳定而可靠的部署，除此之外，在某些情况下，组织一些特定微服务的部署也会增加生态系统的可用性。

如果一个服务无法满足 SLA（参见第 2 章），那么根据该服务的宕机配额情况，后续的部署将会被延后。例如，如果一个服务的 SLA 承诺 99.99% 的可用性（每个月允许宕机 4.38 分钟），但在一个月内宕机了 12 分钟，那么为了满足 SLA，在接下来的三个月都无法进行新的部署。如果一个服务无法通过负载测试（参见第 5 章），那么在通过负载测试之前，该服务就不能被部署到生产环境。对于那些关键性的业务服务来说，它们的中断将导致整个公司无法正常运作，如果这些服务达不到生产就绪标准就不能被部署到生产环境。

服务依赖

有时候人们之所以采用微服务架构是基于这样的想法：微服务可以被独立构建和运行，并被作为大型系统的独立可替换组件。理论上说确实如此，但在现实中，每个微服务都

会依赖其他服务，也会被其他服务依赖。每个微服务都会收到来自客户端（其他微服务）的请求，这些客户端依赖该服务所提供的 SLA，而该服务也会依赖下游的其他服务来完成实际的工作。

构建和运行生产就绪的微服务要求开发人员做好应对依赖失效的准备，减少依赖失效带来的影响，并保护好自己的服务。理解服务之间的依赖关系，并为依赖失效做好应对准备，这些对于构建稳定可靠的微服务来说是非常重要的。

我们举一个例子来说明这个问题的重要性。假设有一个叫作 receipt-sender 的微服务，它的 SLA 是 4 个 9（对它的上游客户端承诺了 99.99% 的可用性）。receipt-sender 依赖了其他几个微服务，其中一个叫作 customers（一个处理客户信息的微服务），另一个叫作 orders（一个处理客户订单信息的微服务）。而 customers 和 orders 又依赖了其他微服务：customers 依赖了 customer-dependency，orders 依赖了 orders-dependency。customer-dependency 和 orders-dependency 很有可能也依赖了其他微服务，这么一来，receipt-sender 的依赖关系就变得复杂起来。

54

receipt-sender 为了达成它的 SLA，并为它的客户端提供 99.99% 的可用时间，开发团队需要确保下游依赖的 SLA 也必须得到满足。receipt-sender 的 SLA 依赖 customers 的 99.99% 可用时间，而如果 customers 的实际可用时间只有 89.99%，那么 receipt-sender 的可用性就会下降到 89.99%。receipt-sender 的每一个依赖项也都面临着同样的情况，如果依赖关系链里的任何一个服务无法保证它的 SLA，就会影响到其他服务。

稳定可靠的微服务需要减少这种依赖失效（无法保证 SLA 也算一种失效）。可以使用备份、回滚、缓存、替代服务等方式来应对依赖失效。

在对依赖失效做出应对计划之前，服务间的依赖关系需要被整理清楚，并记录到文档里，方便以后进行跟踪。任何一个有可能破坏微服务 SLA 的依赖项都必须出现在架构图里，并记录到微服务文档里（参见第 7 章），还要出现在服务监控仪表盘上（参见第 6 章）。另外，可以通过在整个微服务生态系统内实现一个分布式的跟踪系统，为每个微服务生成依赖关系图，自动跟踪所有的依赖项。

在理清了所有依赖项之后，下一步就可以对每个依赖项进行备份，并准备替代服务、回退方案，或者创建缓存。具体怎么做要看每个服务的实际需要。例如，如果一个依赖项的功能可以通过调用另一个服务端点来实现，那么在该依赖项出现失效时，就可以把请求转发给这个服务。如果在依赖项发生失效时请求可以被放到队列里，那么就要去实现一个队列。另一种处理依赖失效的措施是使用缓存：把所有相关的数据缓存起来以便应对依赖服务失效。

LRU 缓存是最常用的缓存类型，它把所有相关的数据保存到一个队列里。当队列被填满

时，那些没有使用过的数据就会被删除。LRU 缓存实现起来很简单（通常使用一两行代码就可以实现），而且它效率很高（不需要网络调用），性能很好（可以立即返回数据），能有效缓解依赖失效带来的影响。这种缓存被称为*防御性缓存*，它在面对依赖失效时能
55 够有效地保护微服务：把从依赖项那里获得的信息缓存起来，当依赖项失效时，你的服务不会受到影响。当然，没有必要为每个依赖项都使用防御性缓存，但对于那些不可靠的依赖项来说，使用防御性缓存可以有效地保护你的微服务。

路由和服务发现

构建稳定可靠的微服务的另一个要求是要确保服务间的通信是稳定可靠的，也就是说，微服务生态系统的第 2 层（通信层，参见第 1 章）必须在整个生态系统范围内做好流量保护工作。与通信层的稳定性和可靠性(除了网络本身)相关的几个问题分别是服务发现、服务注册和负载均衡。

我们应该从主机层面和服务层面了解微服务的*健康状况*。*健康检测*应该一直处于运行状态，避免将请求发送到一个已经中断的主机或服务上。在一个独立通道上运行健康检测可以确保它不会受到其他因素的影响，比如网络堵塞。不过对于微服务来说，在 */health* 端点上硬编码“200 OK”响应并不是一种很理想的健康检测方式，虽然大多数情况下这种方式具有很高的效率。硬编码的响应消息让我们知道微服务还处在启动状态，但不会告诉我们更多的信息，而我们需要的是更准确的信息。

如果一个服务实例的健康状况出现了问题，那么负载均衡器就不应该把流量路由给它。如果微服务整体处在不健康状态（不管是部分主机还是所有主机），那么在问题得到解决之前，流量都不应该被路由到该服务上。

不过，健康检测不应该成为判断一个服务是否健康的唯一手段。有很多未处理的异常也会导致服务被标记为不健康，对于这些情况，应该使用*回路断路器*。如果一个服务出现反常的错误，断路器可以保证不会有请求被发送到该服务上，直到问题得到解决。稳定可靠的路由和服务发现可以防止生产环境的流量和来自其他微服务的请求流向失效的服务实例。

56 服务和端点的解除

解除微服务或 API 端点也是造成微服务生态系统不稳定不可靠的因素之一，这一点经常被人们忽略。如果一个微服务不再被使用，或者开发团队不再为其提供支持，那么需要小心地将它解除，确保不会影响到任何客户端。解除微服务 API 端点是一件很常见的事情：在加入新特性或移除旧特性时，端点经常会发生变更，客户端需要做出相应的更改，

把原先发送到旧端点上的请求切换到新的端点上（或者整体移除）。

在大多数微服务生态系统里，解除服务涉及组织层面的问题，而不仅是技术问题，所以解决起来相当困难。当解除一个微服务时，它的开发团队需要通知所有的客户端并告诉他们如何应对依赖变更所带来的影响。如果被解除的微服务被另一个新的服务代替，或者原有的功能被迁移到另一个已有的服务上，那么它的开发团队需要帮助所有客户端做出相应的更改，以便把请求发送到新的端点上。端点的解除遵循相同的流程：通知客户端，并为他们提供新的端点，或者告诉他们如何应对解除端点所带来的影响。监控扮演了很关键的角色：端点需要被紧密地监控起来，因为在服务或端点被彻底解除之前，还需要检查是否还有请求被发送到已经过时的服务或端点上。

反过来说，不恰当地解除服务或端点会对微服务生态系统造成灾难性的影响。实际发生这种情况的几率要比开发人员认为的要高得多。在一个包含了成百上千个微服务的生态系统里，开发人员在不同团队之间流转，优先级在发生改变，微服务和技术在更新换代。如果没有对这些旧服务或旧技术进行更新或监管，那么一旦有错误发生就不会被注意到，况且这些错误或许在很长一段时间内都得不到解决。如果任由微服务自生自灭，那么一旦它出现中断，就会影响到它的客户端。对于这类微服务，应该将它们解除，而不是对它们置之不理。

对于一个微服务来说，最具破坏性的事情莫过于依赖项的完全失效。没有什么比突如其来的依赖项失效更能造成微服务的不稳定和不可靠了，即使其他团队已经把依赖项失效纳入他们的计划里，也无法避免这种情况的发生。以稳定且可靠的方式解除服务和端点是如此重要，我们无须再次强调。

评估你的微服务

为了帮助读者对他们的微服务和微服务生态系统的生产就绪情况进行有效评估，本书的每一章在章末尾部分都会提供一个简短的问题列表，这些问题跟当前章节所提及的生产就绪标准有关。每个问题就是一个主题，并跟当前章节的每个部分相对应。

开发周期

- 是否有一个可以存放所有代码的中心代码仓库？
- 开发人员所在的开发环境是否准确反映了产品状态（例如，是否准确反映了实际情况）？
- 是否有代码检查、单元测试、集成测试和端到端的测试？
- 是否有代码审查流程和策略？

- 是否具有自动化的测试、打包、构建和发布流程？

部署管道

- 微服务生态系统是否有一个标准化的部署管道？
- 部署管道里是否有 full staging 或 partial staging 阶段？
- staging 环境对生产环境有怎样的访问权限？
- 部署管道里是否有 canary 阶段？
- canary 阶段是否有足够的时间来捕捉所有的缺陷？
- canary 阶段是否准确地模拟了生产环境的业务流量？
- canary 和生产环境的服务端口是一样的吗？
- 生产环境的部署是一步到位还是循序渐进的？
- 对于紧急情况，是否存在直接跳过 staging 和 canary 阶段的情况？

服务依赖

- 微服务的依赖项都有哪些？
- 微服务的客户端都有哪些？
- 微服务如何缓解依赖失效所带来的影响？
- 对于每个依赖项，是否都有备份、替代服务、回退方案或防御性缓存？

58

路由和服务发现

- 微服务的健康检测可靠吗？
- 健康检测是否准确地反映微服务的健康状态？
- 健康检测是否运行在独立的通道上？
- 是否使用了回路断路器来防止不健康的微服务发出请求？
- 是否使用了回路断路器来防止生产环境的业务流量被发送到不健康的主机或服务上？

服务和端点的解除

- 是否有解除微服务的相关流程？
- 是否有解除微服务 API 端点的相关流程？

第 4 章 59

伸缩性和高性能

生产就绪的微服务具备了伸缩性和高性能。一个具备了伸缩性和高性能的微服务以效率作为驱动，它能够高效地处理大量的并发任务和请求，并为未来任务和请求的增长做好准备。这一章将探讨微服务的伸缩性和高性能所涉及的主要组件，包括质的增长规模和量的增长规模、硬件的效能、资源需求和瓶颈的识别、容量的规划、业务流量的伸缩、依赖项的伸缩、任务的处理，以及数据存储的伸缩。

关于微服务伸缩性和高性能的原则

追求高效能是现实世界大型分布式系统架构最为关注的方面，而微服务生态系统也不例外。独立系统（例如单体应用）的效能是很容易被量化的，但在一个大型的微服务生态系统里，有成百上千个小型的服务共同处理任务，要对整个系统的效能进行评估或追求更高的效能会变得非常困难。计算机架构和分布式系统的潜在法则也限制了大型分布式系统的效能：系统越是趋于分布式，系统里包含的微服务越多，微服务个体给整个系统带来的效能就越有限。于是，能够提升系统整体效能的标准化原则就变得很有必要。伸缩性和高性能这两个生产就绪标准，可以帮助我们提升整体系统的效能，并增加微服务生态系统的可用性。

从对微服务和生态系统产生的影响来看，伸缩性和高性能这两个因素之间有着紧密的联系。正如我们在第 1 章所看到的那样，为了构建可伸缩的应用，我们需要考虑并发和分区：60
通过并发，任务被分成更小的单元，而通过分区，这些更小的单元可以被并行处理。所以，伸缩性事关如何对任务进行分而治之，而高性能事关如何衡量应用程序的效能。

在一个快速发展的微服务生态系统里，业务流量在持续增长，每个微服务都需要通过横向扩展来解决性能问题。为了确保微服务的伸缩性和高性能，我们对微服务提出了一些要求。我们需要了解质的增长规模和量的增长规模，这样我们就可以提前为之做好准备。

我们需要合理地利用硬件资源，知道资源的瓶颈和需求，并做好恰当的规划。我们需要确保微服务的依赖项也能随之伸缩。我们需要用可伸缩性和高性能的方式管理好业务流量。我们需要以高性能的方式处理任务。还有最后一点，我们需要以可伸缩的方式存储数据。

一个生产就绪的微服务是可伸缩且高性能的

- 明确的质的增长规模和量的增长规模。
- 高效地使用硬件资源。
- 已识别出资源的瓶颈和需求。
- 容量规划自动化，并通过调度作业来执行。
- 依赖项也会随之伸缩。
- 可以随着客户端的伸缩而伸缩。
- 业务流量模式有章可循。
- 在发生故障时业务流量可以被重新路由。
- 使用支持伸缩性和高性能的编程语言来实现。
- 以高性能的方式处理任务。
- 以可伸缩和高性能的方式存储数据。

了解增长规模

（从上层）了解微服务的伸缩方式是构建和维护可伸缩微服务的第一步。微服务的增长
61 规模包含两个方面的内容，它们是理解和规划微服务伸缩性的基础。第一个是质的增长规模，它反映的是微服务在整个生态系统中所起的作用，以及它将影响到的业务度量指标。第二个是量的增长规模，正如它的名字所表示的，它反映的是微服务能够处理的业务流量的大小。它可以被量化，并有着清晰的定义。

质的增长规模

微服务的增长规模一般是通过微服务所能支持的每秒请求数（RPS）或者每秒查询数（QPS）来描述的，然后通过预测微服务未来的 RPS 或 QPS 得出微服务的增长规模。“每秒请求数”一般用于描述微服务，而“每秒查询数”一般用于描述基于数据或为客户端提供数据的微服务，不过在大多数情况下，这两者是通用的。这些术语很重要，不过要是脱离了上下文环境（特别是微服务在整个生态系统中所处的位置），它们就变得毫无意义。

在大多数情况下，微服务所能支持的RPS或QPS是由初次计算增长规模时的服务状态来决定的：如果只是通过当前业务流量和当前流量负载来计算增长规模，并由此来预测微服务未来的业务流量，这样会存在一定的风险。我们有一些办法可以跳过这个坑，比如负载测试（使用更高的负载对微服务进行测试）。负载测试可以更准确地反映微服务的伸缩性，而且通过分析历史流量数据可以知道流量是如何增长的。不过这里有一个很关键的点没有提到——微服务不是独立运行的，它们是整个生态系统的一部分，这是微服务架构与生俱来的属性。

质的增长规模正是在这个点上起到了作用。微服务的伸缩性通过质的增长规模与上层的业务度量指标联系在一起：微服务可能随着用户量的伸缩而伸缩，也可能随着手机应用使用者数量的伸缩而伸缩，也可能随着订单数量（比如送餐服务）的伸缩而伸缩。这些度量指标和质的增长规模与整个系统和产品有关系，而不仅仅是微服务个体。在业务层面，只有组织层才知道这些度量指标将如何随着时间的推移而变化。当工程团队在理解这些上层的业务度量指标时，开发人员可以把它们与相应的微服务对应起来：如果他们有一个微服务用于处理订单流程，那么通过与未来订单数量相关的度量指标，他们就可以知道他们的微服务将会迎来怎样的业务流量。

当我问起微服务开发团队关于他们服务的增长规模时，他们一般会回答“它每秒可以处理x个请求”。而接下来我会把问题集中在他们的服务将如何在整个产品里发挥作用：请求是什么时候发起的？是不是每次只有一个请求？用户每次在打开应用时是否都会发起一个请求？每次新用户注册产品时是否都会发起一个请求？如果这些问题都得到了圆满回答，那么就可以确定增长规模了。如果请求数量跟打开手机应用的使用者数量有直接关系，那么我们就可以通过预测这些使用者的数量来伸缩服务。如果请求数量是由订餐人数决定的，那么服务的伸缩就可以通过送餐数量来决定，我们就可以通过预测未来送餐数量来伸缩服务。

质的增长规模原则也有例外的情况，越是深入服务栈，就越难确定质的增长规模。内部工具深受这些复杂性的影响，如果它们不可伸缩，那么就会成为业务临界区，组织的其他部分很快也将面临伸缩性挑战。我们不能简单地把服务的增长规模与业务度量指标(例如用户、应用启动量等）监控或告警平台相提并论，平台和基础设施部门需要为他们的用户（开发人员、服务等）和用户的特定需求确定更准确的增长规模。内部工具可以随着部署数量、服务数量、日志数量或者其他数据量而伸缩。因为预测这些数字存在困难，所以事情会变得更加复杂。不过在服务栈的上层，这些数字就变得愈加直接和愈加可预测。

量的增长规模

增长规模的第二个方面是数量，包括RPS或QPS以及其他类似的度量指标。确定量的增长规模要以质的增长规模为前提：量的增长规模是对质的增长规模进行量化的结果。例如，如果质的增长规模是通过“应用启动量”来衡量（有多少人打开了手机应用程序），并且每次应用启动都会发起两个请求以及一个数据库事务，那么每秒请求数和每秒事务数就会成为决定服务伸缩性的关键数字。

我们要准确地选择质的增长规模和量的增长规模，其重要性无须再次强调。我们很快将会看到，在预测服务的运营成本、硬件的需求和限制时，将会用到增长规模。

63

资源的有效利用

在考虑大型分布式系统的伸缩性时，比如微服务生态系统，我们可以把硬件和基础设施资产当作资源来看待。在我们所处的世界里，CPU、内存、数据存储和网络就是资源：它们数量有限，是真实世界里的物理实体，它们是分布式的，生态系统里的各种关键参与者共同分享着这些资源。正如我们在第1章的“组织的挑战”小节所讨论的那样，硬件资源昂贵，有时候很稀缺，在微服务生态系统里存在激烈的资源竞争。

组织层面的资源争用问题可以通过为关键性业务分配更多的资源来缓解。将微服务按照它们在生态系统里的重要程度和对整体业务所产生的价值大小来分类，并根据分类来确定它们对资源需求的优先级：在分配资源时，关键性的业务服务将优先获得稀缺资源。

不过，在实际分配资源时会存在一些困难，因为在微服务生态系统的第1层（硬件层）有太多的决定性因素需要考虑。我们可以为微服务分配专用的硬件资源，每个主机只运行一个服务，但这样做成本太过高昂，而且也不是一种有效利用硬件资源的方式。大部分工程组织选择在多个微服务间共享硬件资源，每个主机上运行多个不同的微服务。实践证明，在大多数情况下，这是一种有效利用硬件资源的方式。

共享硬件资源的风险

在一台机器上运行多个不同的微服务（也就是在微服务间共享机器）通常是对硬件资源的有效利用，不过要确保微服务之间的隔离性，避免它们对其他微服务的性能、效率或可用性造成影响。通过容器（Docker）进行资源隔离有助于防止微服务对其他服务造成伤害。

在微服务生态系统里分配资源最有效的方式之一是对主机进行彻底隔离，比如使用Apache Mesos这样的资源隔离技术对其进行隔离。在这一层面进行资源隔离可以做到动态分配资源，而且可以避免微服务生态系统在资源分配方面存在的很多缺陷。

资源感知

64

在微服务生态系统里有效地分配硬件资源之前，需要先识别出每个微服务对资源的需求和使用瓶颈。资源需求是运行微服务所需要的具体资源（CPU、内存，等），识别资源需求是运行可伸缩微服务的重要前提。资源使用瓶颈是微服务在伸缩性和性能方面存在的限制，它们取决于硬件资源的特性。

资源需求

微服务的资源需求是微服务在运行、处理任务、横向或纵向扩展时所需要的硬件资源。CPU 和内存（在多线程环境里，线程将是第三重要的资源）是最为重要的两种硬件资源。先确定微服务的资源需求，然后具体量化运行一个微服务实例所需要的 CPU 和内存。这对资源隔离、资源分配和决定微服务的整体伸缩性和性能来说至关重要。

识别额外的资源需求

CPU 和内存是最为常见的两个资源需求，不过要注意微服务也需要其他的资源。比如，数据库连接或者应用平台资源（如日志配额）。关注微服务的这些需求有助提升它们的伸缩性和性能。

评估微服务对具体资源的需求是一个漫长的过程，因为有很多相关因素需要考虑。正如之前提到的，最为关键的一点是如何确定单个服务实例所需要的资源。对服务进行扩展最有效的方式是进行横向扩展：随着业务流量的增长，我们可以增加更多主机，并在这些主机上部署服务。要想知道需要增加多少台主机，我们需要先知道服务在单台机器上的运行状况：它能够处理多少业务流量？它需要使用多少 CPU？使用多少内存？有了这些数字，我们就可以准确地知道这些微服务的资源需求。

资源瓶颈

65

通过识别资源瓶颈，我们可以发现和量化微服务的性能和伸缩性限制。微服务在使用资源时会出现资源瓶颈，它们会限制应用的伸缩性。它们可能是基础设施的瓶颈，或者服务架构里的其他一些因素，它们会阻碍应用的伸缩。例如，当数据库的连接数达到极限时，它就会成为微服务的瓶颈。在面临业务流量增长时，微服务需要进行纵向伸缩（而不是进行横向伸缩，横向伸缩通过增加更多的实例或硬件来实现），这也是另一个常见的资源瓶颈：如果只能通过为每个实例增加更多的资源（更多的 CPU 和更大的内存）来实现微服务的伸缩，那么伸缩性的两个原则（并发和分区）就不再适用。

有些资源瓶颈很容易被识别出来。如果你只能通过把微服务部署到拥有更多 CPU 和更大

内存的机器上来满足业务流量增长的需求，说明你的微服务正面临着伸缩瓶颈。遵循并发和分区原则对它们进行重构，让它们可以横向伸缩，而不是纵向伸缩。

纵向伸缩的缺陷

纵向伸缩对微服务架构来说并不是一种可取的伸缩方式。如果每个微服务有自己专门的硬件，或许可以使用这种方式。但在使用硬件隔离技术的今天，比如Docker和Apache Mesos，这种方式不再适用。在构建可伸缩应用时要注意遵循并发和分区原则。

有些资源瓶颈不容易被发现，通过对服务进行负载测试是发现这些瓶颈的最好手段。负载测试将在第5章的“弹性测试”小节进行详细介绍。

容量规划

构建可伸缩微服务最重要的一点是确保微服务在伸缩时能够访问到它们所需要的硬件资源。当微服务需要应对更多的业务流量时，如果没有足够的硬件资源，那么有效使用资源、规划增长规模和从无到有设计可伸缩的微服务这些举措很快会变得毫无意义。对于需要横向伸缩的微服务来说，它们更容易面临这样的挑战。

66 这个潜在的问题不仅带来技术上的挑战，还会让工程组织面临更大的组织层面和业务层面的问题：硬件资源太昂贵，业务和开发团队有预算限制，这些预算（一般会包含硬件预算）需要事先做好规划。为了确保在业务流量增长时微服务能够适当地伸缩，我们可以进行预定的容量规划。容量规划的原则很简单：事先确定每个微服务所需要的硬件资源，把它们加入预算，并确保所需要的硬件能够到位。

我们可以通过增长规模（质和量）、关键性业务度量指标、业务流量预测、已知的资源瓶颈和需求，以及微服务的历史流量数据来确定微服务的硬件需求。这个时候，质的增长规模和量的增长规模会变得很有用，借助它们，我们可以准确地预测微服务的伸缩行为。例如，如果我们知道微服务是根据整体产品的独立访问者数量来伸缩，并且知道每个独立访问者每秒发出的请求数量，还知道公司对下一个季度新增独立访问者数量的预测是20,000，那么我们就可以知道下一个季度我们的微服务需要多大的容量。

每个开发团队、工程组织和公司应该把这些需求加入到预算里。在最终确定预算之前，以预定的方式进行规划可以确保工程组织始终有硬件资源可用，因为预算还没有完全板上钉钉。这里要特别注意（从工程和业务角度），不恰当的容量规划可能带来的成本问题：硬件的短缺会导致微服务无法伸缩，进而导致整个生态系统可用性下降，然后出现系统停运，最终公司要为此付出代价。

新硬件资源的到位时间

开发团队在进行容量规划时容易忽视一个潜在的问题，微服务所需要的硬件资源在规划之时可能并不存在，所以在它们就位之前需要进行申请、安装和配置。在进行容量规划之前，要注意硬件资源的到位时间，避免等待太长时间，而且还要为整个流程预留一些时间，因为中间可能出现延误。

微服务所需要的硬件资源一旦到位，容量规划阶段也就结束了。至于在容量规划之后何时以及如何分配硬件资源取决于每个工程组织以及他们的开发团队、基础设施团队和运维团队。 67

容量规划是一项困难的人工活。与软件工程里的其他人工活一样，它会引入新的失效风险：人工计算可能会暂停，哪怕是一个很小的暂停都有可能给关键性业务服务带来灾难性的影响。将容量规划的主要部分进行自动化可以排除潜在的错误和故障，而这种自动化可以通过使用微服务生态系统应用平台层的自助服务工具来实现。

依赖项的伸缩

微服务的伸缩性受到依赖项伸缩性的制约，如果依赖项无法随着它一起伸缩，那么不管将它构建得多么具有伸缩性，最终仍然会面临伸缩性问题。哪怕只有一个关键性的依赖项无法随着它的客户端伸缩，整个依赖关系链都会受到影响。确保微服务的所有依赖项都可以随着它的预期增长规模进行伸缩是构建生产就绪微服务的前提。

这个问题事关每个微服务以及微服务生态系统里的每一个参与者，这意味着微服务团队需要确保他们的服务不会成为客户端的伸缩瓶颈。换句话说，这给微服务生态系统的其他部分带来了额外的复杂性。来自微服务客户端的业务流量增长是不可避免的，所以需要预先为其做好准备。

质的增长规模和依赖项伸缩

在处理复杂的依赖关系时，只要确保把服务的伸缩问题与上层的业务度量指标关联在一起（使用质的增长规模），那么微服务就能够按照预期的增长规模进行伸缩——尽管跨团队的沟通会有点困难。

依赖项的伸缩问题是实现微服务生态系统伸缩性和高性能的关键所在。大部分微服务都不是独立存在的，几乎每一个微服务都是复杂依赖关系链的一部分。在大多数情况下，整个产品、组织和生态系统的伸缩性要求系统的每一个部分都能够随着系统的其他部分一起伸缩。如果一个系统里只有一小部分微服务具备了高性能和伸缩性，而其他部分无 68

法达到同样的标准，那么达到标准的那部分微服务也变得毫无意义。

除了让微服务开发团队遵循伸缩标准之外，微服务团队之间也要加强合作，确保每个依赖关系链都能够共同伸缩。当业务流量出现增长时，要通知依赖项开发团队。这个时候，跨团队的沟通和协作就会变得很重要：与客户端和依赖项团队定期沟通微服务的伸缩性需求、状态和可能出现的瓶颈，有助于确保参与服务的各方为伸缩做好准备，并识别出潜在的伸缩瓶颈。在这方面，我建议团队间通过架构评审和伸缩性评审会议来达到沟通的目的。在这些会议上，我们会谈到每一个微服务的架构和各自的伸缩性极限，然后一起讨论如何对所有的服务进行伸缩。

流量管理

随着服务的不断伸缩，每个服务所要处理的请求数量也在增长，一个具备高性能和伸缩性的服务必须知道如何处理不断增长的业务流量。在管理业务流量时，需要做到如下几个方面：首先，使用增长规模（质和量）来预测未来流量的增长（或削减）；其次，了解流量模式，并为之做好准备；第三，处理流量的增长，包括流量高峰。

我们已经在这一章的前面部分介绍了第一点：通过了解微服务的增长规模（质和量）可以知道当前流量的负载，并为未来的流量增长做好准备。

了解当前流量模式有助于在底层做好与服务之间的交互。在识别了流量模式之后，包括每秒请求数和所有的关键性度量指标（第 6 章会有更多关于关键性度量指标的介绍），接下来就可以事先安排好服务变更、运维时间和部署，从而避开流量高峰，并且减少新缺陷和重启微服务可能带来的服务中断。根据流量模式对流量进行紧密的监控，并根据流量模式谨慎地调整监控阈值，可以尽早捕捉到问题，避免它们造成服务中断或降低系统的可用性（第 6 章将详细介绍生产就绪微服务的监控原则）。

69 在我们能够预测未来的流量增长并了解当前和过往的流量模式之后，也就知道了流量模式将如何随着预期的增长而变化，接下来我们就可以对我们的服务进行负载测试，确保它们在处理更大的流量负载时能够有预期的表现。第 5 章的“弹性测试”一节会详细介绍负载测试的内容。

流量管理的第三个方面有点复杂。微服务处理流量的方式应该是可伸缩的，也就是说，它们需要为流量的急剧变化做好准备，特别是对于爆发式增长的流量，它们需要谨慎处理，避免整个服务被流量拖垮。不过说起来容易做起来难，在流量发生突然增长时，就算是那些具备了高性能和伸缩性的微服务也会遇到监控、日志和其他问题。我们需要在

基础设施层面做好准备工作，包括所有的监控系统和日志系统，开发团队需要把它们作为弹性测试的一部分。

我想再提一点有关如何管理跨地域业务流量的问题。很多微服务生态系统并不是部署在同一个位置，同一个数据中心，或者同一个城市，它们会横跨全国范围内（甚至全球）的多个数据中心。数据中心经历大规模服务中断的情况并不少见，而当数据中心发生服务中断时，整个微服务生态系统也随之中断。基础设施层（特别是通信层）需要为多个数据中心的流量分布和路由负起责任，不过微服务本身也要做好准备，因为流量会被重新路由。

任务处理

生态系统里的每个微服务都要处理某一类任务。也就是说，每个微服务将会接收来自上游客户端的请求。这些客户端可能需要微服务为它们提供某些信息，或者要求微服务为它们计算或处理某些事情，并把处理结果返回。微服务需要处理这些请求（一般是要求自己的下游服务处理这些任务）并把处理的结果返回给客户端。

编程语言的限制

微服务处理任务的方式多种多样，它们可能直接执行计算或者与下游的服务进行交互。任务的处理与微服务所使用的编程语言和服务的架构（大多数情况下是由编程语言决定 70
的）有关。例如，一个使用 Python 开发的微服务就有很多种高效处理任务的方式，有的需要用到异步框架（比如 Tornado），有的需要用到消息队列，比如 RabbitMQ 和 Celery。所以，一个微服务在处理任务方面的伸缩性和性能在一定程度上与所选择的编程语言有关。

注意编程语言在伸缩性和性能方面的局限性

有很多语言并不是为微服务架构的伸缩性和高性能而准备的，或者它们并没有相关的框架可以满足微服务伸缩性和高性能的要求。

因为编程语言可能会给微服务处理任务的能力带来限制，所以在微服务架构里，编程语言的选择变得非常重要。对于很多开发人员来说，微服务架构的一个优势是可以使用任何一种语言来开发微服务，这个是对的，不过要注意：在选择编程语言时也要考虑它们的约束，而且不要把语言是否流行或者是否有趣（或者开发团队是否对该语言比较熟悉）作为选择的标准，而真正需要考虑的是语言的性能和伸缩性局限。对于微服务开发来说，

没有“最好”的编程语言，只有更合适的语言。

高效地处理请求任务

除了编程语言的选择之外，生产就绪标准要求微服务具有高性能和可伸缩性，也就是说，微服务要能够同时高效地处理大量任务，并为未来的任务增长做好准备。微服务开发团队需要回答如下三个问题：他们的服务是如何处理任务的？他们的服务处理任务的效率如何？当请求数量增长时他们的服务将如何应对？

为了确保伸缩性和高性能，微服务需要高效地处理任务。为了做到这一点，需要并发和分区。并发要求服务不能使用单个进程来处理所有的任务，因为这种方式每次只处理一个任务，并按照一定的次序完成各个步骤，然后再处理下一个，这是一种低效的处理方式。通过并发，每个任务可以被拆分成多个更小的单元。

71

为微服务选择支持并发和分区的编程语言

有些编程语言适合用来高效地（并发和分区）处理任务。在开发新的微服务时，要确保编程语言不会给微服务带来伸缩性和性能方面的限制。对于已经受到编程语言限制的微服务来说，可以选择更合适的编程语言来重写它们——这是一个耗时的任务，不过这样会大幅度提升伸缩性和性能。例如，如果你打算使用一个异步框架来提升微服务的伸缩性和性能，那么选择 Python（而不是 C++、Java 或 Go，要知道，这三种语言擅长并发和分区）将会给你的服务带来很多伸缩性和性能方面的瓶颈，而且后续难以消除。

在对任务进行拆分以后，通过分区可以更高效地处理任务，因为任务可以被并行处理。如果我们有大量的小型任务，可以同时处理它们，只要把它们分配给一组能够并行处理任务的工作线程。如果要处理更多的任务，可以增加更多的工作线程，而不会影响系统的整体性能。所以，并发和分区确保了微服务的伸缩性和高性能。

可伸缩的数据存储

微服务对数据的处理也必须具备伸缩性和高性能。微服务存储和处理数据的方式很容易成为伸缩性和性能的关键影响因素：选择了错误的数据库、错误的 schema 或者一个不支持测试租赁的数据库，到最后会破坏微服务的整体可用性。与微服务其他方面的问题一样，为微服务选择合适的数据库也很复杂，这一章只能讨论个大概。在接下来的章节，我们将会讨论在为微服务生态系统选择数据库时需要考虑的几个因素，并介绍微服务架构在数据库方面所面临的挑战。

微服务生态系统的数据库选择

72

在大型的微服务生态系统里构建、运行和维护数据库不是一件简单的事情。有些采用微服务架构的公司允许开发团队自己选择和维护数据库，有些公司则会选择至少一种可以满足公司大部分需求的数据库，并组建一个独立的团队来运行和维护这些数据库，这样开发团队就可以专注在微服务的开发上。

如果我们能够从微服务架构的 4 个层次（第 1 章的“微服务架构”小节里提到了更多的细节）来考虑问题，而且根据反康威定律，组织架构反映了他们的产品结构，那么我们就可以知道谁应该为数据库的选择、构建、运行和维护负起责任：应用平台层把数据库作为服务，并面向微服务团队开放，而微服务团队使用数据库，并把数据库作为服务的一部分，所以应用平台层和微服务层要负起这个责任。我在现实中见过不同的公司分别采用了这两种方案，它们的效果各有伯仲。我还注意到有一种做法的效果最好：在应用平台层把数据库作为服务，如果这些服务无法满足个别微服务团队的特定需求，那么他们可以使用自己的数据库。

最常见的数据库有*关系型数据库*（SQL、MySQL）和*NoSQL 数据库*（Cassandra、Vertica、MongoDB，以及键值存储引擎，比如 Dynamo、Redis 和 Riak）。在决定使用关系型数据库还是 NoSQL 数据库，并选择一个能够满足微服务需求的特定数据库时，需要考虑以下几个问题：

- 微服务每秒传输的事务数是多少？
- 微服务要存储的是什么类型的数据？
- 微服务需要怎样的 schema？它的变更频率是怎样的？
- 微服务要求的是强一致性还是最终一致性？
- 微服务是读密集的还是写密集的，还是两者兼顾？
- 需要对数据库进行横向或纵向扩展吗？

不过，不管数据库最终是由应用平台团队还是微服务团队来维护，都要先回答上述的几个问题。如果数据库需要进行横向扩展，或者读写会并行发生，那么就需要选择 NoSQL 数据库，因为关系型数据库在横向扩展和并行读写方面不具备优势。

73

微服务架构在数据库方面面临的挑战

微服务架构在数据库方面面临着几个挑战。当多个微服务共享数据库时，就会出现资源竞争问题，有些服务会比其他服务使用更多的可用存储空间。工程师需要为共享数据库设计好伸缩方案，在服务需要更多存储空间时可以横向扩展，以免把可用空间耗尽。

看紧数据库连接

有些数据库对数据库的连接数有严格限制，所以要确保在使用完数据库连接后将其关闭，避免对服务和数据库的可用性造成破坏。

微服务经常面临的另一个挑战是，在进行端到端测试、负载测试和 staging 测试时如何处理测试数据，特别是当服务处在开发阶段和进入部署管道阶段之后。在第 3 章的“部署管道”小节已经提到，部署管道的 staging 阶段会对数据库进行读写。如果选择了 full staging，那么整个 staging 阶段都会使用自己的独立数据库，不过如果选择了 partial staging，那么就需要对生产环境的数据库进行读写，所以在处理测试数据时要十分谨慎：测试数据（也就是所谓的测试租赁处理）需要被清晰地标记出来，并定期清理掉。

评估你的微服务

为了帮助读者对他们的微服务和微服务生态系统的生产就绪情况进行有效评估，本书的每一章在章末尾部分都会提供一个简短的问题列表，这些问题跟当前章节所提及的生产就绪标准有关。每个问题就是一个主题，并与当前章节的每个部分相对应。

增长规模

- 微服务的质的增长规模是怎样的？
- 微服务的量的增长规模是怎样的？

资源的有效利用

- 微服务是运行在专门的硬件上还是共享的硬件上？
- 是否使用了资源隔离技术？

资源感知

- 微服务的资源需求是怎样的（CPU、内存等）？
- 每个微服务实例能够处理多少流量？
- 每个微服务实例需要多少 CPU？
- 每个微服务实例需要多少内存？
- 微服务还需要其他的资源吗？
- 微服务的资源瓶颈在哪里？

- 微服务是否需要被横向或纵向扩展，或者两者兼顾？

容量规划

- 容量规划是否基于调度进行？
- 新硬件多久能够到位？
- 申请硬件的频度是怎样的？
- 是否根据优先级为微服务分配硬件？
- 容量规划是自动化还是手工操作的？

依赖项的伸缩

- 微服务的依赖项有哪些？
- 这些依赖项是否具备了伸缩性和高性能？
- 依赖项能否随着微服务进行伸缩？
- 依赖项的所有者是否做好随微服务进行伸缩的准备？

75

流量管理

- 是否很好地了解了微服务的流量模式？
- 是否根据流量模式来安排服务的变更？
- 流量模式的急剧变化（特别是流量爆发）是否被小心地处理了？
- 在服务失效以后，流量是否能够被恰当地重新路由到其他数据中心？

任务处理

- 微服务所使用的编程语言是否具备伸缩性和高性能？
- 微服务在处理请求时是否存在伸缩性和性能方面的限制？
- 微服务在处理任务时是否存在伸缩性和性能方面的限制？
- 微服务团队的开发人员是否了解他们的服务是如何处理任务的，处理任务的效率是怎样的，以及当任务和请求数量增加时他们的服务将会如何应对？

可伸缩的数据存储

- 微服务是否以可伸缩和高性能的方式处理数据？
- 微服务需要存储什么类型的数据？
- 微服务的数据需要怎样的 schema ？

- 每秒需要处理多少事务？
- 微服务需要更高的读写性能吗？
- 微服务是读密集、写密集还是两者兼顾？
- 微服务的数据库可以横向或纵向扩展吗？它是可复制或者可分区的吗？
- 微服务使用的是专门的还是共享的数据库？
- 微服务是如何存储和处理测试数据的？

第5章 容错和灾备

生产就绪的微服务具有容错和灾备能力。微服务会经常发生故障，在它们生命周期的某些时间点上，任何潜在的故障场景都有可能发生。为了保证微服务生态系统的可用性，我们需要预测故障，并做好灾备工作。为了确保微服务具备优雅的恢复能力，我们需要主动地让服务发生故障，以便验证它们是否能够优雅地从故障中恢复。

这一章将会探讨如何排除故障点，介绍常见的故障场景、如何探测故障并进行补救、如何实现不同类型的弹性测试以及如何在组织层面处理服务事故和中断。

用于构建具有容错能力微服务的原则

在构建大规模分布式系统时，个体组件会经常性地发生故障，没有一个微服务生态系统能够逃逸于这个规则之外。在微服务生命周期的某些时间点上，任何潜在的故障都有可能发生，而且生态系统复杂的依赖关系链会让这些故障变得更加糟糕：如果一个服务发生故障，那么它上游的所有客户端都会受到牵连，整个系统的可用性也会遭到破坏。

让生态系统里的每一个微服务都具备容错和灾备能力是减少灾难损失和避免整体系统可用性受到破坏的唯一途径。

要构建具备容错和灾备能力的微服务，第一步就要排除故障点。生态系统或微服务里不应该存在这样的故障点，它们会导致整个系统或服务不可用。识别出微服务和整个服务抽象层可能发生故障的地方，可以有效避免大部分故障的发生。

接下来是识别故障场景。微服务故障并非都是显而易见的单点故障。容错和灾备要求微服务能够禁受得起内部故障（微服务自身的故障）和外部故障（发生在生态系统其他层的故障）的考验。从主机故障到整个数据中心的故障，从数据库到分布式任务队列，能够让微服务发生故障的因素太多了，而且它们的数量会随着微服务和整个生态系统复杂

度的增加而增长。

在移除了故障点并识别出大部分故障场景之后，接下来就要针对这些场景进行测试，验证微服务能否优雅地从这些故障场景中恢复过来，从而知道微服务是否具备了弹性。微服务的弹性可以通过代码测试、负载测试和混沌测试来验证。

这一步非常关键：在一个复杂的微服务生态系统里，仅仅排除故障是不够的，当服务组件发生故障时，即使最好的挽救措施也有可能变得毫无用处。在生产环境主动地制造故障，不断地对每个组件进行随机的故障测试，这是构建具备容错能力微服务的唯一途径。

不是所有的故障都能够被预测到，构建具有容错和灾备能力的微服务的最后一步需要整个组织的努力。故障检测和挽救措施需要在微服务团队间进行标准化，对于新出现的故障，需要把它们加入到弹性测试里，避免同样的故障再次发生。微服务团队需要训练有素地处理故障：处理服务中断和事故（不管什么级别的）需要在整个工程组织内进行标准化。

一个具备容错和灾备能力的生产就绪的微服务

- 没有故障点。
- 所有可能的故障场景都已被识别出来。
- 已经通过代码测试、负载测试和混沌测试保证了它的弹性。
- 自动化的故障检测和挽救措施。
- 微服务开发团队和整个组织具有标准化的事故和中断处理流程。

避免单点故障

潜在的故障场景首先会出现在每个微服务自身的架构里。如果微服务架构里存在这样的一个点，它的失效会导致这个微服务失效，那么我们就把它叫作微服务的单点故障。虽然微服务里不应该存在这样的点，但事实上这样的情况却经常发生。真实世界的微服务通常不仅存在单点故障，它们还存在多点故障。

例子：消息代理单点故障

为了更好地理解真实生产环境中的单点故障，我们假设有一个 Python 微服务，它使用 Redis（作为消息代理）和 Celery（作为任务处理器）来处理分布式任务。

我们假设 Celery 工作线程（处理任务的线程）因为某些原因无法正常工作，它们也就无法处理任务。这种情况不一定就是单点故障，因为 Redis 可以在 Celery 工作线

程恢复处理能力后进行重试。虽然 Celery 工作线程无法正常工作，但 Redis 仍然在正常运行，任务被堆积到 Redis 的队列里，等待 Celery 工作线程恢复处理能力。不过这个微服务承担了很大的流量（每秒接收几千个请求），任务队列开始膨胀，填满了整个 Redis 服务器。在你意识到问题之前，Redis 的内存已经被耗尽，任务消息开始发生丢失。这种情况已经很糟糕了，不过比这个更糟糕的是，如果多个服务共享硬件，那么使用这个 Redis 作为消息代理的其他服务也开始丢失它们的任务消息。

这就是一个单点故障的例子，也是我在识别微服务单点故障时经常碰到的真实场景。

80

如果微服务发生单点故障，我们可以很容易地识别出来，并将其修复。不过等待微服务发生故障不是一个好主意，我们应该和微服务开发团队一起评审微服务架构，让开发人员把他们的架构图画在白板上，并问他们一个问题“如果架构里的某个点发生了故障该怎么办？”（第 7 章的“理解微服务”一节有更多关于架构评审和识别单点故障的相关内容）。

没有孤立的故障点

微服务生态系统里的服务之间存在复杂的依赖关系链，单个服务的故障点经常会演变成整个依赖关系链的故障点，在极端情况下还会变成整个生态系统的故障点。所以在微服务生态系统里不存在孤立的故障点，识别和排除故障点对于构建具有容错能力的微服务至关重要。

在单点（或多点）故障被识别出来以后，需要将它们排除，如果有可能，需要把它们从架构里移除。如果故障点可以被一些具有容错能力的东西所取代，那么问题就会迎刃而解。可惜我们无法避免每一个故障，还有一些很明显的故障点是我们无法移除的。例如，如果我们的工程组织强制要求使用某些技术，这些技术对于其他开发团队来说很有用，唯独对某个服务会造成单点故障，那么对于这个服务来说就无法彻底移除故障点。为了让服务具备容错能力，只能尽量采取措施来降低故障所带来的负面影响。

故障场景

复杂的大型分布式系统存在这样一个现象：系统总会发生故障，在系统生命周期的某些时间点上，任何可能发生的故障总会不期而遇地发生。

微服务本身就是复杂的系统，它们作为大型分布式系统的组成部分，无法逃逸于这个规则之外。从微服务的 RFC（request for comments）起草之时，直至微服务退役之日，任何可能发生的故障都会不期而遇地发生。灾难无时无刻不在发生：数据中心的机架故障、HVAC 系统故障、生产环境数据库的误删除（这个发生的频率比开发人员承认的要频繁得多）、摧毁整个数据中心的自然灾害。可能发生的故障就真的会发生：依赖项会失效、

81

服务器会失效、软件包会遭到破坏或丢失、监控会失效、日志会丢失（犹如消失在空气里）。

在识别、规避、移除（如果有可能）了微服务架构中的明显故障点之后，下一步需要识别出那些*潜在的故障场景*。我们可以把这些故障场景分为 4 类，并根据它们在生态系统中所在的层来区分。最常见的故障场景如下：硬件故障、基础设施（通信层和应用平台层）故障、依赖项故障和内部错误。在接下来的小节中，我们将进一步介绍这些常见的故障场景，不过先让我们来看一下那些会影响微服务生态系统的常见故障。

首先需要声明，这里列出的潜在故障场景并不能覆盖所有的情况。这里只是要展示一些常见的场景，让读者对他们的微服务或生态系统将会发生哪一种故障有一个清晰的认识，然后引导读者去深究本书其他相关的章节。如果能够遵循本书所述的生产就绪标准（并根据实际情况实现自己的需求），很多故障都可以被规避掉，所以我在这里只介绍一部分故障，并不包含已经在其他章节中介绍过的故障。

常见的生态系统故障

故障会发生在微服务生态系统的每一个层上。这些故障通常都是因缺乏组织层面的标准造成的，它们一般都是运营方面的问题（不一定都是技术问题）。这些故障被归为“运营”故障，但并不代表它们就不如技术故障来得重要或危险，也并不代表这些问题就只能在技术领域之外解决，更不是说开发团队就完全没有责任。这些故障一般都很严重，有些还会影响到技术，这也反映了工程团队之间的不协调。这些常见的故障包括不充分的架构评审、不完善的代码审查、糟糕的开发流程和不稳定的部署过程。

不充分的系统和服务架构评审会导致糟糕的服务设计，在复杂的微服务生态系统里就更是如此。原因很简单：没有一个工程师或者一个开发团队知道基础设施的所有细节，也就无法知道生态系统 4 个层次的复杂性。在设计新系统或架构新服务时，为了保证系统或服务的容错能力，需要让生态系统各个层的工程师参与到整个设计流程中，一起决定如何构建和运行整个复杂的生态系统，这点是至关重要的。不过，就算在系统或服务的设计之初做到了这一点，微服务生态系统在未来仍然会快速演化，在一两年之后基础设施可能变得面目全非，所以让来自组织各个层面的专家定期对架构进行评审，可以确保系统或服务保持最新状态。关于架构评审的更多细节可以参看第 7 章。

不完善的代码审查是另一个常见的故障。虽然这个问题并不只出现在微服务架构里，但微服务架构会加剧这个问题的发生。微服务加快了开发速度，开发人员除了要写好自己的代码，还被要求对其他人的代码进行审查，一天可能需要重复多次。他们还要参加各种会议，为了能让服务运行起来，他们需要完成很多相关的工作。他们需要在不同的任务间切换，在审查代码时很容易忽略一些细节，而写代码的人也是经常没有足够的时间对自己的代码进行审查，代码就直接被部署到了生产环境。这样就给生产环境引入了无

数的缺陷，这些缺陷会造成服务或系统故障，而它们本该在代码审查时被检查出来。有很多方法可以解决这个问题，不过在一个快速迭代的开发环境中，这个问题不可能得到完美的解决。我们需要小心翼翼地对系统和服务进行测试，在代码变更进入生产环境之前需要对它们进行大量的测试。如果在部署时无法捕捉到缺陷，那么最起码要在开发流程或部署管道的其他阶段捕捉到它们。开发流程和部署管道还可能出现另外两个常见的故障。

糟糕的部署是其中之一。“糟糕”的部署一般会包含代码缺陷或失败的构建。糟糕的开发流程和不稳定的部署过程允许故障被带入生产环境，这些故障会让系统或服务以及它们的依赖项发生失效。执行良好的代码审查过程，建立代码审查的工程文化氛围，并给予开发人员足够的时间来审查代码，这是避免故障发生的第一步。不过很多故障仍然难以被捕捉到：如果没有足够的测试，即使再好的代码审查操作也无法准确地预测代码变更将在生产环境发生怎样的行为。只有稳定可靠的开发流程和部署管道才能捕捉到这些故障，防止它们破坏生产环境的系统或服务。第 3 章介绍了如何构建稳定可靠的开发流程和部署管道。

83

总结：常见的生态系统故障

在微服务生态系统各个层面发生的常见故障包括：

- 不充分的系统和服务架构评审
- 不完善的代码审查
- 糟糕的开发流程
- 不稳定的部署过程

硬件故障

硬件处于栈的底层。硬件层包括具体的物理机，基础设施和应用程序代码就运行在这些物理机上。硬件层还包括服务器所在的机架，以及服务器所在的数据中心，如果使用了云服务，还包括服务器的可用区域。硬件层还包括操作系统、资源隔离层、配置管理、主机级别的监控和主机级别的日志（关于硬件层的更多细节，可以回顾第 1 章的内容）。

在这一层会出现很多问题，在这一层发生的灾难性事故（不仅仅是故障）会产生重大影响。这一层是整个生态系统最脆弱的层：如果硬件发生故障而且没有备选方案，那么整个工程组织都会随之崩溃。在这一层发生的灾难性事故一般都是硬件事故：主机崩溃、机架的故障或者整个数据中心的故障。这些灾难性事故比我们所认为的要更经常发生，为了让整个生态系统和每个微服务都具备容错能力，并为这些事故做好应对准备，我们需要有计划地规避这些故障，并做好保护措施。

84 运行在这些机器之上的所有东西都有可能发生故障。机器在投入使用之前需要做好配置，如果配置失败，新的机器就无法投入使用（有时候旧机器也不行）。很多微服务生态系统使用了资源隔离（比如Docker）或资源抽象（比如Mesos和Aurora）技术，这些技术也会出现故障，这些故障会导致整个生态系统中断。糟糕的配置管理或配置变更引起的故障也十分常见，而且难以被检测到。

监控和日志也会发生故障，而且它们一旦发生故障，就无法对中断事故进行诊断，因为诊断所需要的数据不存在。网络也会发生故障（内部和外部的）。最后，关键性硬件组件的停机维护也会导致整个生态系统中断，即使已经在组织层面进行了适当的沟通。

总结：常见的硬件故障场景

部分最为常见的硬件故障场景包括：

- 主机故障
- 机架故障
- 数据中心故障
- 云服务故障
- 服务器配置故障
- 资源隔离或抽象技术故障
- 糟糕的配置管理
- 配置变更引起的故障
- 主机级别的监控故障
- 主机级别的日志故障
- 网络故障
- 停机运维
- 缺乏基础设施冗余

通信层和应用平台层的故障

微服务生态系统的第2层和第3层分别是通信层和应用平台层。这两层处在硬件层和
85 微服务层之间，就像黏合剂一样把整个生态系统桥接在一起。通信层包含网络、DNS、RPC框架、端点、消息系统、服务发现、服务注册和负载均衡。应用平台层包含用于构建和运行微服务生态系统的诸多关键性组件：自助开发工具、开发环境、测试工具、打包工具、发布工具、构建工具、部署管道、微服务级别的日志以及微服务级别的监控。跟硬件层的故障一样，在这两层发生的故障也会影响到整个生态系统，因为开发和运维完全依赖这些组件。让我们来看看在这两层会发生的一些最为常见的故障。

在通信层里，网络故障最为常见。网络故障可以是内部网络（承载了所有的远程过程调用）故障或外部网络故障。另一种网络故障是由防火墙和不恰当的 IP 过滤系统配置造成的。DNS 错误引起的故障也很常见：当 DNS 发生错误时，通信就会中断，而且 DNS 故障难以跟踪和诊断。作为微服务生态系统黏合剂的 RPC 层也是另一个发生故障的源头（也是饱受诟病的），特别是当只有一个通道可以用来连接所有微服务和内部系统的时候。分别为 RPC 和健康检测开通不同的通道会减少故障的发生，因为健康检测使用的通道和处理数据的通道是分开的，不会互相影响。消息系统也有可能发生故障（在之前的 Redis 和 Celery 例子里大致提到过），而且消息队列、消息代理和任务处理器一般都没有备份，对于那些依赖这些组件的服务来说，它们都会成为潜在的故障点。服务发现、服务注册和负载均衡也会发生故障：当它们中的任何一部分出现故障或停机，而且没有备选方案时，那么业务流量就无法被正确分发和路由。

开发流程和部署管道决定了应用平台层将会发生哪些故障，不过不管怎样，这一层也跟生态系统的其他服务一样，会不可避免地发生故障。当开发人员尝试构建新功能或修复缺陷时，开发工具或开发环境可能出现故障。在进行测试、打包、构建和发布时，也会发生类似的故障。如果构建或打出来的包包含缺陷，或者打包出现遗漏，那么部署就会失败。如果部署管道不可用，或者漏洞百出，又或者完全瘫痪，那么部署过程就会中断，不仅新的功能无法部署，对缺陷的修复也无法进入生产环境。最后，微服务的监控和日志也会发生故障，这会导致无法进行问题诊断。

86

总结：常见的通信层和应用平台层故障

部分最为常见的通信层和应用平台层故障包括：

- 网络故障
- DNS 错误
- RPC 故障
- 对请求或响应不恰当的处理
- 消息系统故障
- 服务发现和服务注册故障
- 不恰当的负载均衡
- 开发工具和开发环境故障
- 测试、打包、构建和发布故障
- 部署管道故障
- 微服务级别日志的故障
- 微服务级别监控的故障

依赖项故障

微服务生态系统顶层（微服务层）的故障可以分为两种：一种是由服务内部问题引起的内部故障，一种是由服务依赖项引起的外部故障。我们将首先介绍第二种故障场景。

微服务的下游（也就是微服务的依赖项）出现故障是十分常见的，它们会严重影响微服务的可用性。如果没有做好保护措施，一旦依赖关系链里的一个微服务发生崩溃，就会让所有的上游服务也随之崩溃。当然，一个微服务不一定只是在完全崩溃时才会对上游服务的可用性造成不良影响，如果它无法达到它的 SLA 指标，比如下降了一个 9 或两个 9，那么上游服务的可用性也会随之下降。

87

SLA 不达标的代价

微服务会让上游服务无法达到它们的 SLA 指标。如果一个服务的可用性下降了一个 9 或两个 9，那么所有的上游服务都会受到影响，这要归因于可用性的计算方式：一个微服务的最终可用性是通过自己的可用性乘以下游服务的可用性得出的。无法达到 SLA 指标是一个很严重的（然而却经常被忽视）微服务故障，它会影响到每一个上游服务（依赖关系链往上）的可用性。

其他几种常见的依赖项故障包括：服务调用超时、API 端点的移除（没有恰当地通知上游客户端）和整个微服务的移除。另外，微服务架构并不建议对内部软件包或微服务进行版本管理和切换，因为这样容易出现缺陷，在极端情况下还会发生严重的故障。微服务具有快速演化的特点，内部软件包和服务一直在发生变更，频繁地切换版本（包括软件包和服务的版本）容易导致开发人员使用了不稳定、不可靠甚至不安全的版本。

外部的依赖项（第三方服务或软件包）也会发生故障。外部故障比内部故障更难以检测和修复，因为开发人员基本上无法对它们做任何改动。在微服务生命周期的开始阶段，我们可以适当地降低对第三方服务或软件包的依赖：选择稳定的外部依赖项，或者尽量避免使用外部依赖项，除非完全有必要，这样可以最大程度地避免这些依赖项成为服务的故障点。

总结：常见的依赖项故障场景

部分最为常见的依赖项故障场景包括：

- 下游微服务（依赖项）的故障
- 内部服务中断
- 外部（第三方）服务中断

- 内部软件包故障
- 外部（第三方）软件包故障
- 依赖项未能达到它的 SLA 指标
- API 端点弃用
- API 端点移除
- 微服务弃用
- 微服务移除
- 接口或端点弃用
- 对下游服务调用超时
- 对外部依赖项调用超时

88

内部故障

微服务处在生态系统的顶层。开发团队对这一层出现的故障最为关注，这些故障完全取决于开发和部署的实施情况，以及开发团队以怎样的方式架构、运行和维护他们的微服务。

假设微服务层之下的基础设施足够稳定，那么微服务的大部分事故和中断都是由微服务自身引起的。待命的开发人员发现他们所遇到的大部分问题和故障都是来自他们的服务内部，他们会收到因服务的关键性度量指标发生变更而触发的告警（第 6 章将会介绍更多关于监控、日志、告警和微服务关键性度量指标的内容）。

不完善的代码审查，缺乏适当的测试覆盖，以及糟糕的开发流程（特别是缺乏标准的开发周期），这些因素一般会导致有缺陷的代码被部署到生产环境。在微服务开发团队层面对开发流程进行标准化可以避免出现上述故障（参看第 3 章的“开发周期”一节）。如果没有一个稳定可靠并且包含了 staging、canary 和生产阶段的部署管道来捕捉错误，这些在开发阶段无法被检测出来的错误一旦进入生产环境，它们就会对微服务本身、服务的依赖项以及生态系统里依赖该服务的其他部分造成严重的事故和中断。

与微服务架构相关的组件都会发生故障，包括数据库、消息代理、任务处理系统，等等。代码缺陷和不恰当的异常处理也会造成故障：当微服务发生故障时，未处理的异常和捕捉异常的方式通常会被忽视，而它们却是造成故障的元凶。最后，业务流量的增长也会使服务发生故障，特别是当服务未能为非预期的流量增长做好准备时（关于伸缩性的更多内容，可以参看第 4 章，然后再回到本章后面的“负载测试”一节）。

89

总结：常见的微服务故障场景

部分最为常见的微服务故障场景包括：

- 不完善的代码审查
- 糟糕的架构设计
- 缺乏适当的单元测试和集成测试
- 糟糕的部署
- 缺乏恰当的监控
- 不恰当的异常处理
- 数据库故障
- 伸缩性极限

弹性测试

只是移除架构里的故障点和识别可能出现的故障场景还不足以让微服务具备容错和灾备能力。一个真正具备容错能力的微服务必须能够在不影响自身可用性、客户端可用性和整个生态系统可用性的前提下从故障中优雅地恢复。*弹性测试*是验证微服务是否具备容错能力的最好方式。弹性测试需要主动在生产环境制造所有可能的故障场景，然后不断地让服务随机地发生故障。

一个具有弹性的微服务能够从任何一个故障中恢复，这些故障可以发生在生态系统的任何一个层上：硬件层（例如，主机或数据中心故障）、通信层（例如，RPC 故障）、应用层（例如，部署管道故障）和微服务层（例如，依赖项故障、糟糕的部署或突发的流量增长）。在评估一个微服务的容错能力时，有几种弹性测试可用于确保服务已经为生态系统各个层出现的故障做好了准备。

第一种弹性测试是*代码测试*，它包含了 4 种类型的测试，它们会检查代码的语法、代码
90 的格式、微服务的个体组件、组件间的交互以及微服务如何在复杂的依赖关系链里运作（一般来说，代码测试不属于弹性测试，我在这里把它归为弹性测试是因为：首先，代码测试对于容错和灾备能力来说至关重要；其次，开发团队更倾向于把所有测试都放在一起）。第二种弹性测试是*负载测试*，负载测试会把微服务暴露在较大的流量负载之下，以便验证微服务在流量增长的情况下将会发生怎样的行为。第三种弹性测试是*混沌测试*，混沌测试是一种很重要的弹性测试，微服务将会在生产环境经历主动制造的故障。

代码测试

代码测试是第一种弹性测试，所有的开发人员和运维工程师几乎都对它很熟悉。在微服务架构里，代码测试需要在生态系统的每一个层上进行，包括微服务以及其下的任何一个系统或服务：除了微服务，还有服务发现、配置管理以及其他相关的系统。代码测试也包含了几种类型，包括 *lint* 测试、单元测试、集成测试和端到端测试。

lint 测试

lint 测试可以捕捉代码的语法和格式错误。lint 测试运行在代码上，它们会捕捉与语言相关的问题，可以用于确保代码符合编程语言的风格（有时候还要符合团队或组织的风格）。

单元测试

大部分代码测试是通过单元测试来完成的。单元测试是独立的小型测试，它们运行在微服务的代码“单元”上。单元测试的目的是为了确保微服务的软件部件（例如功能、类、方法等）具备了弹性，并且不包含任何缺陷。不过，大多数开发人员在测试他们的应用和服务时只会想到单元测试。单元测试虽然很好，但还不足以用于评估微服务在生产环境中可能发生的行为。

集成测试

单元测试用于评估微服务的小部件，确保这些部件具备了弹性，而集成测试是第二种代码测试，它会对整个服务进行测试。在集成测试里，微服务的所有小型组件（已经经过单元测试）会被组合在一起进行测试，确保它们可以在一起协同工作。

端到端测试

91

对于一个单体或独立的应用来说，单元测试和集成测试已经足够了，但对于微服务架构来说，微服务和它的客户端以及依赖项之间复杂的依赖关系链给代码测试带来了更高的复杂性，所以需要额外的代码测试来对微服务和它的客户端以及依赖项之间的行为进行评估。也就是说，开发人员需要像在生产环境那样运行端到端测试，这些测试会涉及微服务客户端的端点、微服务自身的端点和微服务依赖项的端点，它们还会向数据库发出查询数据的请求，如果因代码变更出现了问题，它们会在测试过程中被捕捉到。

自动化代码测试

开发团队需要编写以上 4 种代码测试（lint 测试、单元测试、集成测试和端到端测试），不过测试的执行需要在开发周期和部署管理里进行自动化。在开发阶段，在代码变更通过了代码审查之后就要运行单元测试和集成测试。如果代码变更无法通过单元测试或集成测试，那么就不应该被引入部署管道，开发团队需要对它们进行修复。如果代码变更

通过了所有的单元测试和集成测试，那么它们就可以进入部署管道，成为发布到生产环境的候选版本。

代码测试总结

生产就绪的代码测试包含了 4 种类型：

- lint 测试
- 单元测试
- 集成测试
- 端到端测试

负载测试

我们在第 4 章已经看到，生产就绪的微服务必须具备伸缩性和高性能。它需要能够同时处理大量的任务或请求，而且为未来的流量增长做好准备。如果微服务没能为流量、任务或请求的增长做好准备，那么当它们出现增长时，微服务将会出现中断。

作为微服务开发人员，我们知道在未来的某些时刻，微服务的流量会出现增长，我们或许还知道流量将会以一种怎样的形式增长。我们要为流量的增长做好十足的准备，避免出现任何潜在的问题或故障。另外，微服务在达到它们的伸缩极限之前，我们并不了解它们所要面临的伸缩性挑战和瓶颈，所以需要去搞清楚。为了避免出现伸缩性事故和中断，也为了能够为未来的流量增长做好准备，我们可以通过负载测试来测试服务的伸缩性。

负载测试基础

负载测试被用于测试微服务在一定流量负载下的行为。在进行负载测试时，我们会选择一个特定的流量负载，并在微服务上应用该负载，然后在一旁观察微服务的行为。如果微服务在测试过程中出现故障或者其他任何问题，开发人员就可以把这些问题解决掉，否则它们会在未来破坏服务的可用性。

第 4 章提到的增长规模和资源瓶颈可以在负载测试中得到体现。根据微服务的质量增长规模及其相关的业务度量指标，开发团队就可以知道他们的微服务在未来需要处理多大的流量。根据微服务的数量增长规模，开发人员就可以知道他们的微服务每秒需要处理多少个请求或查询。在识别出主要的资源瓶颈和资源需求并解决了资源瓶颈问题之后，开发人员就可以把数量增长规模（以及未来流量的增长）转换成硬件资源规模，有了这些硬件资源，他们的微服务就可以处理更高的流量负载。负载测试可以确保微服务为未

来的流量增长做好了准备。

负载测试也可以用于其他用途，比如用于发现微服务的数量增长规模和质量增长规模，用于发现资源瓶颈和资源需求，用于确保依赖项的伸缩性，用于规划未来的容量需求，等等。如果应用得当，负载测试可以帮助开发人员深入了解微服务的伸缩性（以及伸缩性极限）：负载测试在一个可控的环境里使用特定的流量负载来衡量微服务、微服务的依赖项以及整个生态系统的行为。

在 staging 环境和生产环境运行负载测试

如果能够在部署管道的每一个阶段都运行负载测试，那么它会带来更好的效果。在部署管道的 staging 阶段运行负载测试有一些好处，它可以测试负载测试框架本身，确保流量负载产生了预期的结果，并可以捕捉到可能在生产环境的负载测试中出现的问题。如果部署管道的 staging 阶段使用的是 partial staging，也就是说，staging 环境会与生产环境的服务发生交互，那么要确保负载测试不会影响到生产环境服务的可用性。如果部署管道的 staging 阶段使用的是 full staging，也就是说，staging 环境是生产环境的一个完整镜像，并且 staging 环境的服务不会与生产环境的服务有任何交互，那么就要确保负载测试能够产生准确的结果，特别是当 staging 环境和生产环境之间并不存在主机平价的时候。

不过只是在 staging 环境运行负载测试是不够的。staging 环境可能包含了生产环境的完全镜像和完全的主机平价，但它毕竟不是生产环境。staging 环境不是真实的生产环境，而且 staging 环境很少能够完美地反映生产环境的负载测试结果。在知道了目标流量负载、告知了相关的依赖服务团队，并运行完 staging 阶段的负载测试之后，接下来需要在生产环境运行负载测试。

在进行负载测试时告知相关的依赖服务团队

在运行负载测试时，如果你的服务需要向生产环境的其他服务发送请求，一定要通知这些服务，避免对它们的可用性造成破坏。永远不要假设下游的依赖服务一定能够处理你在负载测试过程中发送给它们的流量负载。

在生产环境进行负载测试有一定的危险性，它很容易导致微服务和它的依赖项出现故障。负载测试是一把双刃剑：大多数情况下，你根本不知道微服务在特定的流量负载下会出现什么行为，也不知道微服务的依赖项会如何处理增长的请求。负载测试的目的是要找出服务的盲点，并确保能够应付预期的流量增长。在把一个服务推向它的极限时，需要有一个自动的熔断机制，一旦出现问题，可以立即停止负载测试。弄清楚微服务的极限并做出相应的调整，在这些调整再次通过测试和部署之后，负载测试可以进入下一步。

94

负载测试自动化

如果组织内的所有微服务（或者一小部分关键性业务的微服务）都需要进行负载测试，而把负载测试的实现全部交由开发团队来设计和运行，这样会引入另一个故障点。理想情况下，应用平台层需要一个自助的负载测试工具或系统，它为开发人员提供一个可信任的、共享的、自动化的集中式服务。

工程组织应该定期运行负载测试，并把它看成组织日常功能的一个组成部分。运行负载测试需要结合流量模式：在生产环境流量处在较低值的时候进行适当的流量负载测试，不要在流量高峰的时候进行测试，以免对服务的可用性造成破坏。如果有一个集中式的自助负载测试系统，那么自动化测试流程就变得非常有用，这个流程可以自动验证测试结果，而且每个服务都可以在上面运行一系列必要的测试。理想情况下，如果这个系统足够可靠，一个无法通过负载测试的微服务就不会被部署。更重要的是，每个负载测试都需要在内部进行记录和公布，这样在测试过程中捕捉到的问题可以很快得到解决。

负载测试总结

生产就绪的负载测试包含以下组件：

- 负载测试使用特定的流量负载，这个负载使用 RPS、QPS 或 TPS 来表示，并通过质量增长规模和数量增长规模计算得出。
- 部署管道的每个阶段都会运行负载测试。
- 负载测试会涉及所有的依赖项。
- 负载测试是完全自动化的，会定期运行，并被记录下来。

混沌测试

在这一章我们介绍了可能在生态系统各个层上发生的各种潜在的故障场景。我们已经知道如何通过代码测试在微服务级别捕捉到潜在的小故障，以及通过负载测试捕捉到伸缩性方面的故障。不过这些测试仍然无法捕捉到隐藏在生态系统其他地方的一些主要故障。

95

为了测试出所有的故障场景，并确保微服务能够从这些故障中优雅地恢复，我们需要另一种弹性测试，也就是混沌测试。

在混沌测试里，微服务会在生产环境经历各种故障，因为一个微服务只有在经历了各种可能发生的故障之后才真正具备了生存能力。每一种故障场景都需要被识别出来，然后在生产环境制造这些故障。通过随机安排这些故障场景可以模拟真实世界复杂的系统故障：一旦开发人员知道系统的某些部分将会发生故障，他们就会为此做好应对准备，不

过这些随机安排的混沌测试会让他们措手不及。

可控的混沌测试

混沌测试必须被严加看管，以免造成整个生态系统的中断。确保混沌测试系统具有适当的权限机制，并且每个事件都需要被记录下来，一旦微服务无法从故障中恢复（或者混沌测试失去控制），解决起来就会容易一些。

与负载测试（以及本书里所提及的其他系统）一样，混沌测试最好可以被构建成服务，而不是指派给开发团队去实现。对测试进行自动化，为每个微服务运行一般性的测试和特定的测试，鼓励开发团队去发现能够让他们的服务发生故障的方式，然后为他们提供资源，基于这些方式来设计混沌测试。确保生态系统的每个部分（包括混沌测试服务本身）都能在一系列混沌测试中存活下来，并不厌其烦地给微服务和基础设施制造压力，直到开发团队和基础设施团队确信他们的服务能够应对各种故障。

不过，并非只有那些使用了云服务的公司才能够进行混沌测试，尽管这些公司最有可能进行混沌测试。在裸机和云服务硬件上发生的故障会有些不一样，不过在云上运行的东西也能在裸机上运行（反过来也可以）。开源解决方案 Simian Army（提供了一个标准的可定制混沌测试套件）适用于大部分公司，不过有特定需求的组织可以构建自己的解决方案。

混沌测试示例

96

部分常见的混沌测试：

- 禁用依赖项的 API 端点。
- 停止所有发送给依赖项的请求。
- 在生态系统的组件间引入延迟，以便模拟网络故障：在客户端和依赖项之间、在微服务和数据库之间、在微服务和分布式任务处理系统之间，等等。
- 停止所有发送到数据中心或区域的流量。
- 随机地关闭主机。

故障检测和修复

弹性测试针对每一个已知的故障场景进行了测试，不过在故障真正发生的时候，我们还需要一个故障检测和修复策略。在介绍能够用于处理事故和中断的流程之前，我们先来看一些技术上的补救策略。

进行故障检测和修复的目的是为了减小故障对用户产生的影响。在一个微服务生态系统里，“用户”可能是另一个微服务（也就是服务的客户端），也可能是产品的真实客户（如果这个服务是面向客户的）。如果故障是由部署引起的，那么最快的解决办法是把服务回滚到上一个稳定的版本。回滚到上一个稳定的版本可以让服务重回已知状态，在这个状态下不会有新出现的故障。对于底层的配置变更也是如此：把配置当成代码来看待，按照不同的版本来部署，如果一个配置变更造成故障，可以很快地回退到上一个稳定的配置。

使用备份是第二种应对策略。如果微服务的一个依赖项失效，可以把请求发送到另一个端点上（如果是端点故障）或者发送给另一个服务（如果是服务故障）。如果请求无法被路由给其他服务或端点，那么需要把它们放进队列或把它们保存起来，直到依赖项的
97 故障被修复。如果数据中心出现故障，需要把流量路由到另一个数据中心。处理故障的方式多种多样，而把流量路由到其他服务或数据中心是最明智的选择：重新路由流量最为简单，而且可以立即减轻对用户产生的影响。

监控是检测故障最有效的手段（第 6 章介绍了关于监控的详尽细节）。检测和诊断系统故障并非人类喜闻乐见的事情，把工程师纳入故障检测流程里只会让他们成为整个系统的故障点。对于故障修复来说也是如此：大部分故障修复都是由工程师来完成的，他们通过痛苦的手动工作来完成修复，却给系统带来了新的故障点，其实我们大可不必这样做。对修复策略进行自动化可以避免潜在的人为错误。例如，如果一个服务在部署之后无法通过健康检测，或者它的关键性度量指标达到了告警阈值，系统可以自动把它回滚到上一个稳定的版本。对于流量的重新路由来说也是如此：如果某些特定的关键性度量指标达到了一定的阈值，可以让系统帮你自动地对流量进行重新路由。容错能力要求系统尽可能地自动移除潜在的人为错误。

事故和中断

微服务和整个生态系统的可用性是标准化的目标。微服务架构的高可用性目标需要通过生产就绪标准及其相关的要求来达成，我所介绍和选择的每一个生产就绪标准也是基于这个目的的。不过这些还不够，为了让每个微服务以及生态系统的每一层都具备容错和灾备能力，开发团队和整个组织需要有适当的处理事故和中断的流程。

任何一次微服务宕机都会降低它的可用性。微服务或生态系统的组件所发生的故障会造成事故或中断，宕机的每一分钟都会影响到它们的可用性，以至于无法达到它们的 SLA
98 指标。大多数公司需要为他们的服务无法达到 SLA 指标和可用性目标付出沉重的代价：服务中断会造成业务上的财务损失，开发团队需要为此承担一部分责任。只要记住这一点，就会知道用于故障检测和修复的时间是如何成为公司财务上的负担的，因为它们影

响的是服务的运行时间（到最后影响的是可用性）。

恰当的分类

并非所有的微服务都同等重要，根据故障对业务的影响程度来对它们进行分类可以简化事故和中断的处理过程。如果一个生态系统包含了成百上千个微服务，每个星期可能会发生数十个或数百个故障。对于一个包含了上千个微服务的生态系统来说，如果百分之十的微服务发生了故障，那么就会有上百个故障。虽说每个故障都需要得到恰当的处理，但并不是说把每个故障都当作紧急事件来处理，然后出动全员待命团队来解决它们。

为了在组织内形成一个持久高效的事故和中断处理流程，需要做两件事情。首先，根据故障对系统的影响程度对微服务进行分类，这样就可以对事故和中断安排优先级（这样也有助于解决资源竞争问题，包括工程资源和硬件资源）。其次，事故和中断也需要被分类，这样人们就可以清楚地知道每个故障的影响范围和严重等级。

微服务的分类

为了缓解资源的争用问题，也为了确保事故能够得到恰当的处理，我们可以（也应该）将生态系统里的微服务按照它们对业务的重要程度进行分类。一开始，只需进行粗略地分类，不要求完美。关键在于要标记出对于业务来说很重要的微服务，它们拥有最高的优先级和影响，其他的微服务则拥有较低的优先级，这取决于它们与关键性业务的紧密程度。基础设施层一般会拥有最高的优先级：硬件层、通信层、应用平台层里的任何一个被关键性微服务所依赖的组件应该具有最高的优先级。

事故和中断的分类

每个事故、中断或故障都可以从两个维度进行衡量：第一个是严重程度，第二个是影响范围。严重程度跟应用程序、微服务或系统的类型有关，如果微服务涉及关键性的业务（例如，产品的一个业务或面向用户的功能离不开它），那么故障的严重程度应该跟服务的类型相匹配。另外，影响范围是指故障对生态系统能够造成的影响范围的大小，它可以被分为三个级别，分别是高、中、低：如果一个事故的影响范围被标识为“高”，那说明这个事故影响的是整个业务或一个外部功能（比如面向用户的功能）；一个影响范围被标识为“中”的事故影响的是服务本身，或者它的一些客户端；一个影响范围被标识为“低”的事故所产生的负面影响可能不会被它的客户端、其他服务或外部用户注意到。换句话说，严重程度应该根据事故对业务的影响程度来划分，而影响范围应该根据事故影响的是局部还是全局来划分。

让我们举几个实际的例子来说明这个问题。我们把每个故障的严重程度分为 4 个级别（0 至 4，0 级最重，4 级最轻），同时使用高、中、低 3 个级别来区分影响范围。先让我们

来看一个简单的例子：一个彻底的数据中心故障。如果一个数据中心全面瘫痪（由于某些原因），那么严重程度必然是 0（它影响的是整个业务），而且影响范围必然是“高”（再次声明，它影响的是整个业务）。我们再来看另一个场景：假设我们有一个微服务，它提供了一个关键性的业务功能，并且发生了 30 分钟的故障，它的一个客户端因此受到影响，不过系统的其他部分并没有受到影响。那么这个故障的严重程度可以为 0（它影响的是关键性的业务），同时影响范围为“中”（它并没有影响整个系统，只影响到自己和它的一个客户端）。最后，我们假设有一个用于生成微服务模板的内部工具，它发生了几个小时的故障，那么我们应该如何对它进行分类？生成微服务模板（并进行微服务轮换）并不是一个关键性业务，而且没有影响到用户功能，所以它的严重程度不应该是 0（或许也不应该是 1 或 2）。不过，因为是服务自身的故障，所以或许我们可以把它的严重程度划分为 3，影响范围是“低”（故障只影响到这个服务本身）。

处理事故的 5 个步骤

当发生故障时，一个标准化的事故处理流程对保证整个系统的可用性来说是至关重要的。当发生事故或中断时，一系列清晰的处理步骤可以缩短诊断和解决故障的时间，而这反过来减少了每个微服务的宕机时间。当今业界在处理一个事故时有一个典型的三步走标准流程：分诊（triage）、缓解（mitigate）和解决（resolve）。不过，如果采用了微服务架构，

为了具备高可用性和容错能力，需要在事故流程里增加两个额外的步骤：协调和跟踪。总的说来，事故处理需要 5 个步骤（如图 5-1 所示）：评估、协调、缓解、解决和跟踪。

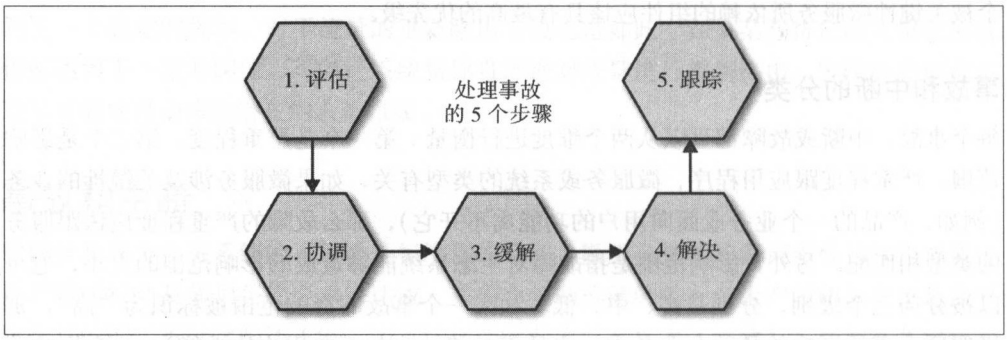


图5-1：处理事故的5个步骤

评估

当服务的关键性指标触发了告警（第 6 章有更多关于告警、关键性度量指标和待命轮班的详细内容），待命的开发人员需要对告警做出响应，他们首先要做的是对事故进行评估。待命工程师是第一批需要对事故做出响应的人，他们需要尽快对每个触发告警的问题进行分诊，并确定问题的严重程度和影响范围。

协调

在对事故进行了评估和分诊之后，下一步需要协调开发团队一起来讨论事故的细节。待命的开发人员不可能解决所有的问题，让相关团队参与进来可以确保问题得到快速解决。所以，事故和中断需要有一个清晰的沟通通道，那些严重程度较高且影响范围较大的问题就可以立即得到足够重视。

事故的沟通内容需要被记录下来，这点很重要。首先，记录沟通过程（对话记录、邮件等）有助于诊断和排查问题：每个人都知道谁做了哪些修复，知道哪些可能的故障根源已经得到排除，并且一旦定位到了问题根源，每个人对问题就有了彻头彻尾的了解。其次，有必要让服务的客户端知道服务发生了故障，尽量减少负面影响，保护好自己。这样可 101 以保持整体的高可用性，避免一个服务拖垮整个依赖关系链。再次，在事故报告中记录事故的严重程度和影响范围时，沟通记录提供了事故发生的细节以及它是如何得到诊断、缓解和解决的。

缓解

第三步是缓解。在问题得到评估并在组织范围内进行过讨论之后（确保找到对的人来解决问题），开发人员需要减轻事故对客户端和业务造成的影响。缓解并不是解决问题，它并没有彻底地修复问题，只是在减小事故造成的影响。只有当一个服务和它的客户端恢复了可用性，才算完成了缓解。

解决

在对事故和中断进行了缓解之后，工程师们就可以着手解决问题，这是事故处理流程的第 4 步。这一步要修复在缓解时还未处理好的问题，SLA 指标的计算会在这个时候暂停。用于计算微服务 SLA 的两个最为重要的时间量分别是检测时间（TTD）和缓解时间（TTM）。在完成缓解之后，事故不会再对用户或服务的 SLA 有任何影响，所以解决问题的时间（TTR）很少（或者永远不会）会被用来计算服务的可用性。

跟踪

事故处理流程的最后一步是跟踪，在这一步需要完成三件事情：事故需要被记录到报告里，以便对它们进行分析，严重的事故需要让更多的人知道，并为开发团队提供一个行动事项列表，他们可以借此让发生故障的微服务重回生产就绪状态（行动事项一般会写在事故报告里）。

记录事故报告是跟踪事故最为重要的部分。一般来说，事故报告会描述事故是怎样发生的、为什么会发生事故以及本该做哪些事情来防止事故的发生。具体地说，事故报告至少需要包含事故概要、事故相关数据（检测时间、缓解时间、解决问题的时间、总宕机

时间、影响到的用户数量、相关的图表等）、详细的时间表、问题根源综合分析、为防止事故发生本该采取的措施、如何防止未来发生类似的事故，以及一个能够让微服务重
102 回生产就绪状态所需要的行动事项列表。一个恰到好处的事故报告是很有价值的，它并不是用来指责任何人，而是记录服务的客观事实。因发生事故而指责开发人员无法让组织从事故中得到教训，而这些对于维护一个稳定可靠的生态系统来说是至关重要的。

在复杂的微服务生态系统里，任何一个能够拖垮某个微服务的故障，不管大小，它也一定能够拖垮系统里的其他微服务。在团队间（以及整个组织里）共享已有故障的信息，有助于将其扼杀于襁褓之中。如果执行得当，事故评审会大幅提升解决问题的效率，我曾经看到过参加事故评审的开发人员在散会后以冲刺般的速度对他们的服务缺陷进行了修复。

评估你的微服务

为了帮助读者对他们的微服务和微服务生态系统的生产就绪情况进行有效评估，本书的每一章在章末尾部分都会提供一个简短的问题列表，这些问题与当前章节所提及的生产就绪标准有关。每个问题就是一个主题，并跟当前章节的每个部分相对应。

避免故障点

- 是否存在单点故障？
- 是否存在多点故障？
- 故障点是否能够被移除，或者需要对它们进行缓解吗？

故障场景

- 是否所有可能的故障场景都已被识别出来？
- 有哪些横跨整个生态系统的常见故障？
- 硬件层有哪些故障会影响到这个微服务？
- 通信层和应用平台层有哪些故障会影响到这个微服务？
- 哪些依赖项故障会影响到这个微服务？
- 哪些内部故障会拖垮这个微服务？

103 弹性测试

- 这个微服务是否通过了适当的 lint 测试、单元测试、集成测试和端到端的测试？
- 这个微服务是否经过合格的负载测试？
- 是否通过混沌测试对所有可能的故障场景进行了测试？

故障检测和修复

- 在组织里是否有标准化的事故处理流程？
- 这个微服务的故障是如何影响业务的？
- 是否对故障进行了清晰的分级？
- 是否有清晰的缓解策略？
- 当发生事故时，团队是否遵循了事故处理的 5 个步骤？

第 6 章 <105>

监控

生产就绪的微服务需要处在恰当的监控之下。为了构建一个生产就绪的微服务，并保证更高的可用性，恰当的监控是非常重要的。这一章将介绍用于微服务监控的主要组件，包括关键性度量指标、如何记录这些指标、构建展示这些指标的仪表盘、触发告警，以及关于待命的最佳实践。

用于微服务监控的原则

微服务生态系统的主要中断事故都是由糟糕的部署引起的，除此以外，缺乏恰当的监控也是一个常见原因。理由很简单，如果我们不了解微服务的状态，如果我们不去跟踪那些关键性度量指标，那么在发生中断之前，那些致命的故障会一直躲在暗处。等到微服务发生了中断，它的可用性已经遭到破坏。用于减灾和修复故障的时间更是让服务的可用性雪上加霜：没有那些关键性指标的信息，开发人员好比面对着一堵空白的墙壁，无法快速地解决问题。所以说恰当的监控非常重要：它为开发团队提供了微服务的所有信息。如果微服务得到恰当的监控，它的状态就会一目了然。

生产就绪的微服务的监控包括 4 个组件。第一个组件是*日志*，日志里包含了重要的信息，开发人员可以从中了解到微服务当前和过去的状态。第二个组件是*仪表盘*，一个设计良好的仪表盘能够确切地反映微服务的健康状况，组织里的每一个人都能够轻松地看懂仪表盘，并了解微服务的健康状况。第三个组件是*告警*，关键性度量指标的告警具有可操作性，可以帮助开发人员解决问题。最后一个组件是为微服务监控建立一个可行的*轮班待命*机制。以上 4 个组件能够为微服务的可用性带来保护作用：故障和错误会被检测出来，并在它们拖垮系统之前得到解决。 <106>

一个得到恰当监控的生产就绪的微服务

- 它的关键性度量指标在主机级别、基础设施级别和微服务级别得到识别和监控。
- 它有能够反映微服务过去状态的日志。
- 它的仪表盘包含了所有的关键性度量指标，而且很容易读懂。
- 它的告警具有可操作性，并且定义了阈值。
- 有一个专门的轮班待命机制负责监控微服务，并对事故和中断做出响应。
- 有一个清晰的、良好定义的标准待命流程，用于处理事故和中断。

关键性度量指标

在深入讲解监控组件之前，首先要知道我们想要以及需要对什么东西进行监控：我们想要监控一个微服务，但这意味着什么呢？微服务并不是一个可跟踪的个体，它不能被隔离起来，它比我们所认为的要复杂得多。在上百个服务器上部署微服务，每个微服务的行为反映的是所有微服务的整体行为，很难简单地对其进行量化。关键是要找出微服务的哪些属性能够有效地描述它们的行为，以及当这些属性发生变化时，它们能够告诉我们关于微服务健康状况的哪些信息。我们把这些属性叫作关键性度量指标。

有两种关键性度量指标：一种是主机和基础设施的度量指标，一种是微服务的度量指标。主机和基础设施的度量指标跟微服务所运行的基础设施设和服务器有关，而微服务的度量指标跟每个微服务个体有关。如果从第 1 章所描述的微服务的 4 层模型来看，主机和基础设施的度量指标属于第 1 层到第 3 层，而微服务的度量指标属于第 4 层。

107

把度量指标分为上述两种类型，从组织层面和技术层面来看，都是非常重要的。受主机和基础设施度量指标影响的通常不止一个微服务：例如，如果一个服务器出现问题，而这个服务器被多个微服务共享，那么主机级别的度量指标就跟所有相关的微服务开发团队相关。同样的，微服务度量指标通常跟整个开发团队有关，而不仅仅是某个人。开发团队需要对这两个度量指标（跟他们的微服务相关的度量指标）进行监控，并且那些跟多个微服务相关的度量指标需要在相关团队间广而告之。

需要得到监控的主机和基础设施度量指标包括：微服务在每台主机上使用的 CPU、内存、线程数、文件描述符，以及数据库连接数。这些度量指标里应该包含基础设施和微服务的信息，也就是说，监控信息的粒度需要细到能够让开发人员知道微服务运行在单个主机和多个主机上的状态。例如，开发人员需要知道他们的微服务在某台主机上使用了多少 CPU，以及在微服务运行的所有主机上使用了多少 CPU。

当主机级别的监控遇上了资源隔离技术

有些微服务生态系统使用了集群管理应用（比如 Mesos），将主机级别的资源（CPU、内存等）进行隔离。在这种情况下，主机级别的度量指标就不可用了，不过微服务团队还是要对微服务的整体度量指标进行监控。

微服务级别的度量指标监控会复杂一些，因为它们跟微服务的开发语言有关。每种开发语言都有特定的任务处理方式，一般来说，这些跟开发语言相关的特性也需要被纳入监控范围。假设有一个 Python 服务，它使用了 uwsgi，那么 uwsgi 的工作线程数就是一个需要被监控的度量指标。

除了跟开发语言特性相关的度量指标外，我们还需要对以下度量指标进行监控：服务的可用性、服务的 SLA、延迟（服务的延迟和 API 端点的延迟）、成功调用 API 端点的次数、API 端点的响应时间和平均响应时间、发送请求（以及请求的目标端点）的服务（客户端），错误和异常（包括已处理的和未处理的），以及依赖项的健康状态。

108

不管应用程序被部署到哪里，所有的度量指标都应该被监控起来。这意味着部署管道的每一个阶段都应该处在监控之下。staging 环境必须被小心地监控起来，这样就可以在新的候选版本进入生产环境之前捕获到它们的问题。在进入生产环境之前的所有部署阶段都需要得到监控，包括 canary 部署阶段和生产环境部署阶段（关于部署管道的更多细节，请参看第 3 章）。

在识别出微服务的关键性度量指标之后，下一步就是要捕捉微服务发出的度量指标。捕捉、记录、图形化展示，然后触发告警。后面我们会介绍如何实现这些步骤。

关键性度量指标总结

主机和基础设施的度量指标

- CPU
- 内存
- 线程
- 文件描述符
- 数据库连接

微服务的度量指标

- 开发语言相关的度量指标
- 可用性

- SLA
- 延迟
- 调用成功的端点
- 端点响应结果
- 端点响应时间
- 客户端
- 错误和异常
- 依赖项

109

日志

日志是第一个监控组件。日志与微服务属于相同的代码库，它们存在于每个微服务里，用于记录微服务的状态信息。事实上，日志的主要目的就是为了能够描述微服务在过去某些时间点的状态。

微服务架构的好处之一是它为开发人员带来了自由度，开发人员可以频繁地部署新的特性和进行代码变更，不过这也导致了微服务一直处于变化的状态：大多数时候，微服务每几个小时就会发生变更，就算是几天变更一次，重现某些问题也变得几乎不可能。当出现问题时，日志是诊断问题根源的唯一途径，用于确定微服务在发生中断时的状态并找出发生中断的原因。开发人员需要通过日志来判断微服务在哪些地方发生了问题以及它们是怎么发生的。

不带版本管理的微服务日志

微服务架构不建议使用版本管理，因为它会促使其他服务（客户端）在各种版本间切换，而有些版本可能不是最好或最新的。没有了版本管理，要确定微服务在发生故障时的状态就会变得很困难，不过详尽的日志让这件事情变得不再是个问题：如果日志信息足够详细，微服务发生中断时的状态就可以很容易地被识别出来，缺乏版本管理不再成为解决问题的障碍。

要在日志里记录哪些信息因每个微服务而异：记录任何能够有效描述当时微服务状态的信息。我们可以通过对微服务代码里的日志进行限制，从而缩小日志的信息范围。应用程序不会（也不应该）记录主机和基础设施的信息，不过运行在应用平台上的服务和工具会记录这些日志。有些微服务级别的度量指标应该被记录在微服务日志里，比如用户ID的散列值、请求消息和响应消息的细节，等等。

当然，有些东西永远不要被记录到日志里。日志里不应该包含任何身份信息，比如客户

的名字、社会保障号，以及其他敏感的私人数据。日志里也不应该包含任何存在安全风险的信息，比如密码、访问密钥。在大多数情况下，即使是那些无关紧要的信息，比如用户 ID 和用户名，也不应该被记录，除非经过加密保护。 110

有时候，仅有微服务级别的日志还不够。微服务不是独立存在的，它们在生态系统里和它们的客户端及依赖项有着复杂的关系链。开发人员尽其所能记录和监控微服务的重要信息，并对贯穿整个关系链的请求消息和响应消息进行端到端的跟踪，可以揭示一些重要的系统信息（比如系统的延迟和可用性），否则这些信息就会被隐藏起来。为了让这些信息可见，需要跟踪每个请求消息在系统里的整个流转过程。

读者或许会注意到，需要被记录到日志里的信息太多了。日志就是数据，记录日志是要付出代价的：存储日志需要成本，访问日志需要成本，跨网络存储和访问日志都需要付出额外的成本。对于个体微服务来说，或许看不出存储日志的成本有多高，不过如果把整个生态系统的日志加起来，就可以知道成本是多么高昂了。

日志和调试

要避免在代码里添加调试日志，因为这些代码如果被部署到生产环境，它们产生的日志是很耗费成本的。如果出于调试目的往代码里添加日志，要确保这些代码不会被部署到生产环境。

日志应该具有伸缩性和可用性，并且可以很容易地被访问和搜索。为了降低记录日志的成本，并确保它们的伸缩性和可用性，需要为服务强制规定日志配额，比如哪些信息可以被记录到日志里，每个微服务可以保存多少日志，以及日志在清理之前可以保存多长时间。

仪表盘

每个微服务至少需要一个*仪表盘*，这个仪表盘会收集并展示所有的关键性度量指标（比如硬件的使用情况、数据库连接、可用性、延迟、响应，以及 API 端点的状态）。仪表盘的图形化界面可以实时地反映微服务的重要信息。仪表盘应该是集中式和标准化的，并且可以很容易被访问到。 111

仪表盘应该简单易懂，这样人们就能够快速地从中看出微服务的健康状况：任何一个人都应该能够立即从仪表盘上看出微服务是否运行正常。这个需要在展示内容上做出权衡：只展示精简的必要信息。

仪表盘应该准确地反映整个微服务的监控质量。仪表盘应该包含所有能够触发告警的度

量指标（后面我们会介绍告警）：如果有任何度量指标被忽略，那么说明监控做得不到位，而如果仪表盘所包含了非必需的度量指标，那么说明告警的设计缺乏最佳实践（最终会导致糟糕的监控）。

不过对于一些例外情况，非关键性的度量指标也可以被包含在仪表盘里。除了关键性度量指标，部署管道每个阶段的信息也应该被展示在仪表盘上，不过并不一定要把它们展示在同一个仪表盘上。因为要监控的度量指标太多，开发人员有可能会选择为不同的部署阶段设置不同的仪表盘（staging、canary 和生产阶段），这样就可以精确地反映每个部署阶段的微服务健康情况：因为不同阶段的部署会同时进行，所以要求仪表盘能够反映各个部署阶段的微服务健康情况（可以把不同部署阶段的微服务看成不同的微服务，毕竟它们是不同的微服务实例）。

仪表盘和故障检测

尽管仪表盘能够反映微服务度量指标的异常现象和负面趋势，但开发人员不应该只通过查看仪表盘来检测事故。这是一种反模式，它会导致糟糕的告警和监控。

把部署时间展示在仪表盘上有助于定位由新部署造成的问题。最有效的方式是把部署时间展示在每个度量指标的图表里，在部署完毕之后，开发人员通过快速检查图表可以看到度量指标的异常情况。

112 设计良好的仪表盘可以帮助开发人员更容易地检测异常和设置告警阈值。度量指标的微小变动有可能会逃过告警，不过一个精准的仪表盘可以帮助开发人员捕捉到这些变化。在下一小节我们将会介绍告警阈值。设置告警阈值不是一件容易的事情，这是众所周知的。不过我们可以结合仪表盘的历史数据来设置告警阈值：开发人员可以从度量指标里看到正常状态和尖刺状态，尖刺状态说明在过去发生过中断（或者导致中断），那么就可以根据这些状态来设置阈值。

告警

实时告警是生产就绪微服务监控的第三个组件。可以通过告警来实现故障检测和度量指标变化的检测。所有的度量指标，包括主机级别的度量指标、基础设施的度量指标和微服务的度量指标，它们都需要被加入告警，并为它们设置不同的阈值。具有可操作性的有效告警对于保证微服务可用性和减少微服务宕机时间来说是非常重要的。

设置有效的告警

所有的度量指标都必须设置告警。主机级别、基础设施级别或微服务级别的任何一个度量指标发生变化，只要它们会造成服务中断、引起延迟或者破坏微服务的可用性，都需要触发告警。如果度量指标发生丢失，也要触发告警。

所有的告警都应该有它们的用处：应该为它们设置合适的阈值。阈值可以分为三种，每一种都有其上下边界：正常（normal）、警告（warning）和严重（critical）。正常阈值反映的是一般性情况，说明度量指标处在正常边界以内，不应该触发告警。当度量指标偏离了基准，警告阈值会触发告警，说明微服务即将出现问题。警告阈值会在度量指标偏离基准进而发生中断之前发出告警，否则只能任其对微服务造成负面影响。严重阈值要基于造成中断或延迟的度量指标边界进行设置。理想情况下，警告阈值应该在度量指标达到严重阈值之前触发告警，以便让待命人员进行快速检测和减灾，并解决问题。每一种阈值的设置都应该高到可以避免无用的告警，同时低到可以捕捉到所有问题。

在微服务生命周期的早期设定阈值

如果没有历史数据，阈值的设定会变得很困难。在微服务生命周期早期设定的阈值会存在一些风险，它们有可能发挥不了作用或者会触发太多的告警。在给新的微服务（或者旧服务）设定阈值时，开发人员可以通过负载测试来估计阈值。使用“正常”的流量来运行负载测试可以得到正常阈值，而使用超预期的流量可以得到警告阈值和严重阈值。

告警应该具备可操作性。不具备可操作性的告警有可能被待命的开发人员忽略，因为它们可能不重要，跟微服务不相关，无法反映微服务的问题，或者它所反映的问题是开发人员无法解决的。如果待命开发人员无法立即对告警采取行动，那么这些告警就不应该出现在告警池里，它们应该被重新分配给其他相关的待命团队，或者（如果可能）把它们变得具有可操作性。

有些微服务的度量指标可能不具备可操作性。例如，关于依赖项可用性的告警可能就不具备可操作性，因为依赖项在发生中断、延迟或宕机时，客户端可能什么都做不了。对于不具备可操作性的度量指标，可以适当设置阈值，或者在极端情况下，可以不设置告警。不过，只要有一点点的可操作性，哪怕事情再小，比如仅仅是联系依赖项的待命开发团队着手解决问题，这样的告警也需要被触发。

处理告警

一旦告警被触发，它们就要被得到快速有效的处理。触发告警的根源需要被找出来并解决掉。遵循以下几个步骤可以快速有效地处理告警。

第一步是为每个已知的告警提供排查步骤，详细说明如何进行诊断、缓解并解决问题。这些排查步骤应该被写进运行手册，放进微服务的文档里，让微服务的待命人员能够方便地查看到它们（更多关于运行手册的内容可以在第 7 章找到）。运行手册对于微服务的监控来说是非常重要的：开发人员可以遵循手册里的排查步骤来对告警进行缓解，并
114 解决问题。触发告警是因为度量指标出现了偏离，运行手册里可以包含度量指标和造成度量指标偏离基准的原因，以及如何对问题进行调试。

有两种运行手册需要被创建。第一种运行手册针对主机级别和基础设施级别的告警，它需要包含所有主机级别和基础设施级别的度量指标，并在整个组织内进行分享。第二种运行手册针对每个具体的微服务，它包含处理告警的排查步骤，例如，延迟的高峰值会触发告警，那么运行手册里就应该有相应的排查步骤，清楚地说明该如何调试、缓解和解决延迟问题。

第二步是识别告警反模式。如果告警已经让微服务待命团队不堪重负，而微服务的工作状态仍然达不到预期，那么那些重复出现但相对简单的告警需要被自动化处理掉。也就是说，可以把解决问题的步骤融合进微服务里。自动化处理加上运行手册里的排查步骤，可以让告警处理更加高效。事实上，任何只需要简单几步就可以处理好的告警都可以被自动化。这一层级的监控一旦被建立起来，微服务就不会重复经历相同的问题。

轮班待命

在微服务生态系统里，开发团队需要对微服务的可用性负责。既然有了监控，那么开发人员就需要为他们的微服务做好随时待命的准备。待命人员的职责很明确：他们要在他们的排班期内检测、缓解和解决微服务问题，避免这些问题造成微服务中断或影响业务。

在一些大型的工程组织里，网站可靠性工程师、DevOps 工程师或其他运维工程师会负责监控和待命，不过前提是微服务需要达到相对可靠稳定的程度。在大多数微服务生态系统里，很少有微服务能够达到如此高的稳定性，因为正如我们之前所见，微服务无时不刻在发生变更。开发人员需要为他们自己部署的代码负起监控的职责。

有一个好的轮班待命机制是非常重要的，它要求整个组织的团队都参与其中。为了避免
115 出现疲劳，轮班待命应该保持简单：每次排班不少于两个开发人员参与，每次排班不要超过一个礼拜，开发人员每个月最多参与一次排班。

轮班信息应该在组织内公开，如果一个微服务的依赖项出现问题，微服务团队可以快速联系到依赖项的开发团队。把这些信息公布在一个统一的地方，有助于开发人员更有效地诊断问题，从而避免服务中断。

在工程组织范围内建立标准化的待命流程将与微服务生态系统的构建过程相伴始终。开发人员应该训练有素，知道如何进行轮班待命，了解待命的最佳实践，并能够快速加入到轮班待命中。建立标准化的待命流程，让每个开发人员明白待命的目标，这样可以避免出现疲劳、疑惑和沮丧。

评估你的微服务

为了帮助读者对他们的微服务和微服务生态系统的生产就绪情况进行有效评估，本书的每一章在章末尾部分都会提供一个简短的问题列表，这些问题跟当前章节所提及的生产就绪标准有关。每个问题就是一个主题，并跟当前章节的每个部分相对应。

关键性度量指标

- 这个微服务有哪些关键性的度量指标？
- 有哪些主机级别和基础设施级别的度量指标？
- 有哪些微服务级别的度量指标？
- 这些关键性度量指标都被监控起来了吗？

日志

- 这个微服务需要把哪些信息记录到日志里？
- 这个微服务是否记录了重要的请求消息？
- 日志是否准确反映了微服务在各个时间点的状态？
- 这个日志方案是否具有伸缩性和高性价比？

仪表盘

116

- 这个微服务是否有仪表盘？
- 仪表盘是否简单易懂？是否所有的关键性度量指标都展示在了仪表盘上？
- 是否能够从仪表盘上看出这个微服务是否运行正常？

告警

- 是否每个度量指标都设有告警？
- 是否所有告警都设置了合适的阈值？
- 告警阈值设置是否恰当，以便在发生中断之前触发告警？
- 告警是否具有可操作性？
- 运行手册里是否包含了用于诊断、缓解和解决问题的排查步骤？

轮班待命

- 是否有一个专门的轮班待命机制用于微服务的监控？
- 每次待命排班是否有至少两个开发人员参与？
- 这个待命流程是否在整个工程组织内进行了标准化？

第7章

文档化和理解

生产就绪的微服务具有良好的文档并且为人所理解。组织蔓延和技术债务是在采用微服务架构时需要面临的两个最为关键的挑战，对微服务进行文档化和理解微服务有助于开发人员在这两方面做出权衡。这一章将探讨微服务文档化和理解微服务的重要元素，包括如何创建详尽的文档、如何在微服务生态系统的每一个层面理解微服务，以及如何在整个工程组织内实现生产就绪。

微服务文档和理解的原则

作为微服务标准化的最后一个原则，我将引用一个著名的俄罗斯文学作品作为开头。尽管在一本介绍软件架构的书里引用 Dostoevsky 的作品看起来有点奇怪，但我找不到比这个更合适的故事，因为它与微服务文档化和理解微服务的精髓是如此契合。

“Rakitka，你要知道，或许我算不上是一个有道德的人，但毕竟我献出了一个洋葱。”

——Grushenka，*The Brothers Karamazov*

Dostoevesky 的 *The Brother Karamazov* 通过作品里的人物 Grushenka 来讲述故事，我最喜欢的部分是关于一个老太太和一个洋葱的故事。故事梗概如下：

从前有一个自私无情的老太太。有一天，她遇到了一个乞丐，因为某些原因，她对乞丐起了怜悯之心。她想给这个乞丐一些东西，不过她只剩下一个洋葱了，于是她把洋葱给了乞丐。最后，老太太死了，因为她生前太刻薄，最终下了地狱。在经历了一番苦难之后，有个天使来救她，因为上帝想起了她生前对乞丐做出的那个善行，打算也对她仁慈一回。天使拿了一个洋葱来到老太太面前，老太太一把抓过洋葱。不过令她感到沮丧的是，她 118
身边的鬼魂也想要她手中的洋葱。此刻，她的冷血本性显露出来，她挣扎着把他们推开，不让他们抢到洋葱。在保护洋葱的过程中，洋葱被撕成了碎片。最后，老太太和其他鬼

魂又跌回了地狱。

这不是一个暖人心房的故事，不过我发现 Grushenka 所讲述的故事与微服务文档化的本质非常契合：把洋葱贡献出来。

为微服务创建详尽的文档并及时更新，其重要性不言而喻。如果你问微服务开发人员他们的主要关注点是什么，他们会支支吾吾地告诉你一堆还没有实现的功能、还没有修复的缺陷、出现问题的依赖项，以及对他们的微服务和依赖项的各种疑问。如果与他们深究微服务和依赖项的细节，他们一般会说：我们不知道它是怎么工作的，它是一个黑盒，而文档毫无用处。

糟糕的依赖项文档和糟糕的内部工具文档会束缚开发人员的手脚，阻碍他们开发出生产就绪的微服务。因为没有合适的文档，开发人员无法正确使用依赖项和内部工具，因此浪费了很多时间。因为没有合适的文档，他们需要通过反向工程来搞清楚服务或工具的工作原理。

糟糕的文档会破坏开发人员的生产力。在缺乏运行手册的情况下，待命人员每次都需要从头开始解决问题。如果没有项目上手指南，新的开发人员需要从头开始理解服务的工作原理。服务的故障点在造成中断之前不会被人注意到。新增的特性与整个服务的大方向相去甚远。

生产就绪的微服务文档是集中式的微服务知识库。知识的分享包含两方面内容：一方面是关于微服务的信息，一方面是组织对微服务的理解。糟糕的文档所带来的问题可以被分为两个子问题：文档的缺乏和理解的缺乏。要解决这两个子问题，需要对微服务进行标准的文档化，并把它们放在合适的地方，方便人们了解微服务。

119 Grushenka 所讲述的故事就是微服务文档化的黄金准则：把洋葱贡献出来。为了自己贡献洋葱，为了那些和你一起工作的同事贡献洋葱，为了那些依赖了你的微服务的开发人员贡献洋葱。

一个生产就绪的微服务具有良好的文档并且为人所理解

- 它有详尽的文档。
- 它的文档会定期更新。
- 它的文档包含如下内容：微服务描述、架构图、待命人员的信息、重要信息的链接、开发上手指南、服务请求消息流、端点的信息、依赖项的信息、运行手册，以及常见问题答疑。
- 它为开发人员、团队和组织所理解。

- 它符合生产就绪标准并且满足相关要求。
- 它的架构经过了反复的评审。

微服务文档

组织内所有的微服务文档都应该被集中放置在一个人们方便访问的公共位置。任何一个开发人员都应该能够轻易地访问到这些文档。在组织里为这些文档建立一个内部站点是最好的办法。

README 文件和代码注释不是文档

很多开发人员把微服务的文档写在 README 文件里，或者写在代码的注释里。README 文件和代码注释固然重要，不过它们都不是生产就绪的文档，而且开发人员需要将代码拉取到本地之后才能对其进行搜索。文档应该被放置在一个中心位置（就像网站一样），组织的所以文档都放置在这里。

文档要定期更新。当服务发生重大变更时，文档也要随之更新。例如，如果新增了一个 API 端点，端点的信息需要被添加到文档里。如果新增了一个告警，排查步骤也需要被添加到运行手册里。如果新增了一个依赖项，这个依赖项的信息也需要被添加到文档里。记住，贡献出你的洋葱。

120

要达到上述要求，最好把更新文档作为开发流程的一部分。如果更新文档被看作独立于开发之外的任务，那么它就有可能不会被执行，还会带来潜在的技术债务。为了减少技术债务，应该鼓励（如果有必要可以要求）开发人员在完成代码变更后对文档进行相应更新。

把更新文档作为开发周期的一部分

如果更新文档被看成编码的附属任务，那么它经常会被推迟，进而成为技术债务。把更新文档作为开发周期的一部分可以避免这个问题。

文档应该详尽，并且包含有用的信息。它应该包含与服务有关的信息。开发人员在阅读了文档之后，可以知道：（1）如何开发服务，（2）服务的架构，（3）轮班待命信息，（4）服务是如何运行的（请求消息流、端点、依赖项，等等），（5）如何诊断、缓解、解决事故和告警，（6）回答经常被问及的服务相关问题。

最重要的是，文档要清晰易懂。包含了太多术语的文档没有多大用处，太过技术化却没有解释与服务相关细节的文档也没有多大用处，没有包含深入细节的文档也没有多大用

处。之所以要求文档清晰易懂，是为了让公司里的所有开发人员、开发经理、产品经理和其他管理人员都能读懂。

接下来让我们深入介绍生产就绪文档所包含的内容。

描述

文档应该以服务的描述作为开头。描述应该简洁并切中要点。例如，假设有一个叫作receipt-sender的微服务，它的作用是在客户完成下单之后发送收据，那么对它的描述应该如下所示。

121

描述：

在客户完成下单后，receipt-sender通过邮件向客户发送一份收据。

这个描述确保任何一个看了这份文档的人都能够明白这个微服务在整个系统中所扮演的角色。

架构图

紧跟在服务描述之后的是架构图。架构图应该包含服务的架构细节，包括组件、端点、请求消息流、依赖项(上游和下游的)，以及数据或缓存信息。图7-1是一个架构图的例子。

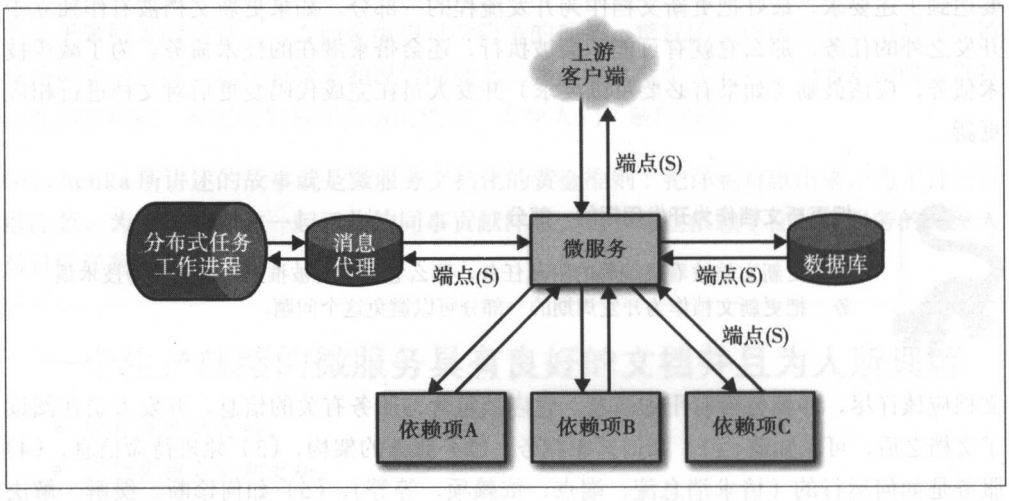


图7-1：微服务架构图示例

架构图有几个重要的用途。如果只是通过阅读代码几乎无法弄清楚微服务的工作原理，好的架构图从视觉上对微服务进行描述和概括。开发人员在添加新特性时，架构图可以

帮助他们抽离出服务内部组件，这样他们就会知道该在哪里以及如何添加新特性。更关键的是，如果没有架构图，有些问题就不会被注意到：只是通过看代码很难看出服务的故障点，但在精确的架构图里，故障点一览无遗。

轮班待命信息

122

能够看到文档的人可能是：(1) 服务开发团队的开发人员或者 (2) 另一个服务开发团队的开发人员，他们正承受服务故障所带来的影响或者想知道服务的工作原理。对于后一种开发人员来说，能够访问到前一种开发人员的相关信息是有必要的，所以文档里需要包含轮班待命信息。

轮班待命信息应该包含待命人员（包括开发人员、开发经理和产品经理）的名字、位置和联系信息。如果其他团队的人对服务有任何疑问或者碰到了服务问题，他们就知道该联系谁。这些信息很有用，当一个开发人员碰到了依赖项问题，他就知道该联系谁，并知道对方在团队中的角色，从而让团队沟通变得简单高效。

在文档里增加轮班待命信息（并保持更新）可以确保人们在遇到一般性问题或紧急情况时知道该联系谁：也就是开发微服务的待命工程师。

链接

文档应该成为所有微服务的信息之源，所以文档里应该包含代码库的链接（这样开发人员就可以拉取代码）、仪表盘的链接、微服务的 RFC 链接，以及最新的架构评审记录。文档里还可以包含其他微服务的信息以及微服务所使用的技术信息，这些信息对于开发人员来说可能会有用，它们也应该被包含在文档的链接部分。

开发上手指南

开发上手指南可以帮助新的开发人员融入开发团队，并开始贡献代码，为微服务添加新的特性，并在部署管道里引入新的变更。

上手指南的第一部分应该是运行微服务的相关步骤。它包含拉取代码、配置运行环境、启动服务，以及验证服务是否运行正常（包含运行微服务的所有命令和脚本）。

上手指南的第二部分需要让开发人员明白微服务的开发周期和部署管道（开发周期和部署管道的相关细节可以分别在第 3 章的“开发周期”和“部署管道”小节里找到）。它需要包含每个步骤的技术细节（例如需要用到的命令，并提供一些例子）：如何拉取代码、如何对代码进行修改、如何编写单元测试（如果有必要的话）、如何运行测试、如何提交变更、如何进行代码审查、如何确保服务被正确地构建并发布，以及如何部署（包括

123

部署管道是如何建立起来的)。

请求消息流、端点和依赖项

文档里还应该包含关于请求消息流、端点和服务依赖项的重要信息。

请求消息流的文档里可以包含请求流程图。如果架构图里已经详细描述了请求消息流，那么可以把架构图放在这里。流程图需要有相应的文字描述，比如请求的类型以及它们是如何被处理的。

这块内容也可以包含微服务所有的 API 端点信息，最好是一个包含了每个端点名字和响应消息描述的列表。这些信息对于其他开发团队的人来说必须足够清晰和易于理解，这样他们在阅读了 API 端点的描述之后就可以把你的服务看成一个黑盒，他们可以顺利地调用端点并收到期望的响应。

这块内容的第三部分是服务依赖项的信息。包括依赖项、依赖项的相关端点、发给依赖项的请求消息、依赖项的 SLA、依赖项的备份（或缓存），以及依赖项的文档和仪表盘链接。

运行手册

在第 6 章中已经提过，每一个告警说明都应该被包含在运行手册里，手册里还应该包含如何诊断、缓解和解决这些告警的排查步骤。运行手册的内容应该被包含在服务文档里，并有独立的章节，包括用于诊断和调试错误的指南，指南既要包含一般性的描述也要有细节的描述。

好的运行手册总是以待命的需求和过程作为开头，然后列出所有告警内容。运行手册需要包含每一个告警的名称、告警的描述、问题的描述，以及用于诊断、缓解和解决告警
124 的排查步骤。运行手册还应该描述告警与组织层面之间的关系：问题的严重程度、是否出现了中断、与其他团队沟通需要哪些信息。

运行手册应该简单易懂

待命的开发人员可能（事实上是经常会）在任何时候接到任务，包括深夜或者凌晨。所以运行手册应该简单易懂，即使是一个带有睡意的开发人员也能轻松地看懂运行手册。

保持运行手册的清晰和简单是非常重要的。待命的开发人员或者正在解决问题的开发人员通过阅读运行手册能够快速地采取行动、诊断问题、缓解事故、解决问题，这些都能够在很短的时间内完成，从而把服务的宕机时间降到最少。

不过，并不是每一个告警都很容易诊断和解决，有很多的中断（除了新部署所引入的代码方面的缺陷）是之前没有遇到过的。为了让开发人员能够处理这些问题，需要在运行手册里添加诊断和调试章节，这些内容可以帮助开发人员系统地解决新出现的问题。

问答章节

文档的问答章节包含了有关服务的常见问题及其回答，而这一章常常被人们忽略。把常见的问答写入运行手册，可以减轻待命人员和团队的负担。

问答章节应该包含两类问题。第一类是由其他团队的开发人员提出的问题，如果有人问你一个问题，而你认为这个问题会被再次问到，那么就可以把它加到问答章节里。第二类问题是由本团队成员提出的问题，任何有关服务的问题都可以被加入问答章节，比如如何做一些事情、为什么这么做、什么时候做。

总结：生产就绪微服务的文档需要包含的内容

125

生产就绪微服务的文档需要包括：

- 关于微服务以及微服务在整个生态系统中所扮演角色的描述。
- 一个能够详细描述微服务架构及其客户端和依赖项的架构图。
- 微服务开发团队的待命信息。
- 代码库链接、仪表盘链接、RFC 文档链接、架构评审文档链接，以及其他任何有用的相关信息的链接。
- 开发上手指南，包含开发流程、部署管道，以及任何能够帮助开发人员开发上手的有用信息。
- 微服务请求流程、SLA 指标、生产就绪状态、API 端点、客户端和依赖项的详细信息。
- 运行手册，包含一般性的事故处理流程和用于诊断、缓解、解决告警问题的分步操作指南，以及如何排查和调试问题。
- 问答章节。

理解微服务

除了详尽的集中式文档之外，还需要组织层面的流程来确保微服务可以被每一个开发团队和整个组织所理解。不管从哪方面来讲，一个满足了生产就绪要求的微服务，那它一定能够为人所理解。

对于开发人员、团队和整个组织来说，理解微服务是非常有必要的。乍一看，“理解”微服务这样的说法无法清晰地表达其含义，不过我们可以借助生产就绪微服务的概念，在系统的各个层面对微服务的理解进行定义。有了生产就绪标准和要求，以及对组织复杂
126 性和微服务架构挑战的真实理解，开发人员就可以把对每一个微服务的理解进行量化，并为组织做出贡献。

对于开发人员来说，他们可以从容地回答有关他们微服务的问题。例如，当被问及他们的微服务是否具备了伸缩能力时，他们可以检查一下伸缩性要求列表，然后自信地回答“是”或者“不是”，或者其他中间答案（例如，“它满足了X和Z，但还不满足Y”）。类似的，当被问及他们的微服务是否具备容错能力时，他们可以列举出所有的故障场景，然后详细地解释他们是如何通过各种弹性测试为这些故障场景做好应对准备的。

对于团队来说，理解微服务意味着他们已经知道他们的微服务与生产就绪标准还有多少距离，以及该如何把他们的微服务带向生产就绪状态。为了能够成功地到达这个状态，理解微服务应该成为组织的文化：生产就绪标准和要求应该成为团队决策的推动力，它们不仅仅是一个检查列表，更是指引团队构建良好微服务的原则。

对微服务的理解需要在整个组织内部生根发芽，生产就绪标准和要求需要成为组织流程的一部分。在构建微服务之前，需要发出RFC（request for comments）请求评审，人们根据生产就绪标准和要求对微服务进行评估。开发人员、架构师和运维工程师对微服务的方方面面进行评估，包括稳定性、可靠性、伸缩性、性能、容错能力、灾备能力、监控、文档和理解，在微服务正式进入生产环境之前，确保它具备了可用性，并可以放心地让它进入生产环境。

只是在微服务生命周期的开始阶段应用生产就绪标准是不够的。我们需要持续地对已有的微服务进行评审，让每个微服务的质量可以保持在一个很高的水平，并在各个微服务团队间和整个生态系统内具有高可用性。对已有服务的审计进行自动化，并在组织内部公开审计结果，这样可以让整个组织了解微服务系统的整体质量情况。

架构评审

在对上千个微服务和它们的开发团队实施了生产就绪标准和要求之后，我意识到架构评审是促进微服务理解的最有效手段。架构评审就是一个会议，所有的开发人员和网站可
127 靠性工程师（或者运维工程师）坐在一起，在白板上画出服务的架构图，然后对其进行评审。

在几分钟之内，开发人员和整个团队对评审的内容就会有很清晰的了解。通过架构评审，开发人员能够快速发现伸缩性和性能方面的瓶颈、之前未曾发现的故障点、未来可能出

现的中断和故障场景，并知道需要新增哪些特性。之前糟糕的架构决定在这个时候就会原形毕露，需要被替换掉的旧技术也会显现出来。为了确保架构评审的效率和目的性，可以让其他团队（特别是基础设施、DevOps 或网站可靠性工程团队）的开发人员也加入进来，他们经历过大规模分布式系统的架构，能够指出一些容易被忽略的问题。

每次会议应该生成一个最新的服务架构图和一个在接下来的几周或几个月内需要完成的项目列表。新的架构图也需要被加到文档里，而项目可以被包含到服务的线路图（见后面的“生产就绪路线图”一节）和 OKR（objectives and key results）里。

微服务演化得很快，开发速度也很快，微服务生态系统的底层持续地发生变化。为了确保服务架构能够得到很好的理解，评审会议需要定期举行。我发现最好的做法是把评审会议的安排与 OKR 和项目计划结合在一起。如果项目和 OKR 是按照季度来安排的，那么架构评审会议也应该每季度举行一次，而且是在项目计划开始之前。

生产就绪审计

为了确保微服务符合生产就绪的标准和要求，微服务团队可以对服务进行*生产就绪审计*。审计很简单：针对生产就绪的标准和要求列表，检查服务是否满足所有要求。这样有助于理解微服务：在审计结束之后，开发人员和他们的团队就会知道他们的服务所处的位置并做出相应的改进。

审计的结构需要反应生产就绪的标准和要求。开发团队需要通过审计对微服务的各个方面进行量化：稳定性、可靠性、伸缩性、容错能力、灾备能力、性能、监控，以及文档。之前已经提到过，这里的每一个标准都包含一系列要求，开发人员需要通过调整这些要 128
求来达成组织的目标和需求。这些要求的具体内容视不同公司的微服务生态系统而定，不过标准和基本组件是相通的（参见附录 A）。

生产就绪路线图

微服务开发团队通过审计，对他们的服务是否达到生产就绪标准有了清晰的了解，接下来要计划如何将他们的服务带向生产就绪的状态。之前的审计工作在这个时候起到了作用：开发团队已经知道他们的服务在哪些方面还达不到生产就绪的要求，接下来的工作就是针对这些方面做出改进。

这个时候可以引入*生产就绪线路图*，而且我发现线路图在整个生产就绪和理解微服务流程里可以起到非常大的作用。微服务各不相同，每个微服务达到生产就绪要求的实现细节也不一样，详细的线路图可以帮助团队朝着生产就绪目标前进。需求里可以包含实现的技术细节、已经出现过的问题（中断和事故）、任务管理系统的 ticket 链接，以及开发

人员的名字。

线路图可以作为项目计划和OKR的一部分。向生产就绪状态迈进的过程最好可以与新特性的开发和新技术的采用齐头并进。让生态系统里的每一个微服务都具备稳定性、可靠性、伸缩性、高性能、容错能力、灾备能力、监控能力、文档化，并为人们所理解，这是确保微服务在满足生产就绪标准最直接最有效的途径，同时保证了整个生态系统的可用性。

生产就绪自动化

架构评审、审计和路线图解决了开发人员和开发团队在理解微服务时所面临的挑战，不过要在组织层面理解微服务，还需要更多的组件。到目前为止，我所描述的所有用于构建生产就绪微服务的工作内容都需要人工参与，开发人员需要遵循每一个审计步骤，制订任务、列表和路线图，并检查每一项需求。太多的人工参与容易带来技术债务，即使是在最有效率的团队里也是这样。

软件工程最主要的原则之一：如果你需要不止一次地手动完成同一件事，那么就将它自动化。这个原则也可以应用在运维任务上，包括一次性的紧急情况以及任何需要在命令行终端输入命令的场景。当然，它同样适用于在工程组织范围内推动生产就绪标准。自动化就是你能够奉献给团队的洋葱。

为每个微服务制订生产就绪要求列表是一件简单的事情。我在Uber是这么做的，我也看到其他一些公司的开发人员也在这么做。于是我创建了一个列表模板（参见附录A），可供读者参考。这个列表可以简化生产就绪的自动化。例如，为了检查容错能力和灾备能力，可以运行自动化检查任务，确保每个微服务通过了适当的弹性测试。

生产就绪检测的自动化的困难程度完全取决于服务的复杂程度。如果所有微服务和自助服务工具都提供了完备的API，那么实现自动化就很简单。而如果服务之间的通信存在问题，或者自助服务工具使用起来很烦琐或者功能缺失，那么实现自动化就会是一件很困难的事情（不仅难以达到生产就绪标准，服务和整个生态系统的完备性也会受到影响）。

生产就绪的自动化在多个方面促进了组织对微服务的理解。如果能够实现自动化并持续运行这些检测任务，开发团队就能够持续地了解每个微服务的状态。在组织内部公开检测结果，为每一个微服务打分，提高关键性服务的最低标准分，严格控制部署过程。生产就绪应该成为工程文化的一部分，这样才能保证最终达成生产就绪。

评估你的微服务

为了帮助读者对他们的微服务和微服务生态系统的生产就绪情况进行有效评估，本书的

每一章在章末尾部分都会提供一个简短的问题列表，这些问题与当前章节所提及的生产就绪标准有关。每个问题就是一个主题，并与当前章节的每个部分相对应。

微服务文档

130

- 微服务的文档是否被集中存放在一个公开的、人们容易访问到的地方？
- 文档是否方便搜索？
- 微服务发生重要变更时，文档是否也得到相应的更新？
- 文档是否包含了微服务的描述？
- 文档是否包含了架构图？
- 文档是否包含了待命信息？
- 文档是否包含了重要信息的链接？
- 文档是否包含了开发上手指南？
- 文档是否包含了微服务请求消息流、端点和依赖项的信息？
- 文档是否包含了运行手册？
- 文档是否包含了问答章节？

微服务理解

- 团队里的每个开发人员是否都能回答与他们的微服务的生产就绪相关的问题？
- 微服务是否遵循了一系列原则和标准？
- 对于新的微服务，是否有 RFC 流程？
- 已有的微服务是否经常得到评审和审计？
- 是否每个微服务团队都举行架构评审？
- 是否有生产就绪的审计流程？
- 是否有生产就绪路线图用于把微服务带向生产就绪的状态？
- 生产就绪标准是否推动了组织的 OKR？
- 生产就绪流程是自动化的吗？

生产就绪检查列表

以下检查列表可以用于所有的微服务，可以手动进行也可以自动化进行。

一个生产就绪的微服务是稳定且可靠的

- 它有一个标准化的开发周期。
- 它的代码需要经过初步检查、单元测试、集成测试以及端到端的测试。
- 它的测试、打包、构建和发布流程是自动化的。
- 它有标准化的部署管道，包括 staging 阶段、canary 阶段和生产阶段。
- 它的客户端是已知的。
- 它的依赖项是已知的，而且是有备份的，还有可选的回退方案以及缓存，以防出现依赖项失效。
- 它有稳定可靠的路由和服务发现机制。

一个生产就绪的微服务是可伸缩且高性能的

- 明确的质的增长规模和量的增长规模。
- 高效地使用硬件资源。
- 已识别出资源的瓶颈和需求。
- 容量规划自动化，并通过调度作业来执行。
- 依赖项也会随之伸缩。
- 可以随着客户端的伸缩而伸缩。 <132>
- 业务流量模式有章可循。
- 在发生故障时，业务流量可以被重新路由。
- 使用支持伸缩性和高性能的编程语言来实现。

- 以高性能的方式处理任务。
- 以可伸缩和高性能的方式处理和存储数据。

一个具备容错和灾备能力的生产就绪的微服务

- 没有故障点。
- 所有可能的故障场景都已被识别出来。
- 已经通过代码测试、负载测试和混沌测试，保证了微服务的弹性。
- 自动化的故障检测和挽救措施。
- 微服务开发团队和整个组织具有标准化的事故和中断处理流程。

一个得到恰当监控的生产就绪的微服务

- 它的关键性度量指标在主机级别、基础设施级别和微服务级别得到识别和监控。
- 它有能够反映微服务过去状态的日志。
- 它的仪表盘包含了所有的关键性度量指标，而且很容易读懂。
- 它的告警具有可操作性，并且定义了阈值。
- 有一个专门的轮班待命机制负责监控微服务，并对事故和中断做出响应。
- 有一个清晰的、良好定义的标准待命流程，用于处理事故和中断。

133

一个生产就绪的微服务具有良好的文档并且为人所理解

- 它有详尽的文档。
- 它的文档会定期更新。
- 它的文档包含了如下内容：微服务描述、架构图、待命人员的信息、重要信息的链接、开发上手指南、服务请求消息流、端点的信息、依赖项的信息、运行手册，以及常见问题答疑。
- 它为开发人员、团队和组织所理解。
- 它符合生产就绪标准并且满足相关要求。
- 它的架构经过了反复的评审。

评估你的微服务

为了帮助读者对他们的微服务和微服务生态系统的生产就绪情况进行有效评估，本书的每一章在章末尾部分都会提供一个简短的问题列表，这些问题与当前章节所提及的生产就绪标准有关。每个问题就是一个主题，并跟当前章节的每个部分相对应。

稳定性和可靠性

开发周期

- 是否有一个可以存放所有代码的中心代码仓库？
- 开发人员所在的开发环境是否准确反映了产品状态（例如，是否准确反映了实际情况）？
- 是否有代码检查、单元测试、集成测试和端到端的测试？
- 是否有代码审查流程和策略？
- 是否具有自动化的测试、打包、构建和发布流程？

部署管道

- 微服务生态系统是否有一个标准化的部署管道？
- 部署管道里是否有 full staging 或 partial staging 阶段？
- staging 环境对生产环境有怎样的访问权限？
- 部署管道里是否有 canary 阶段？
- canary 阶段是否有足够的时间来捕捉所有的缺陷？
- canary 阶段是否准确地模拟了生产环境的业务流量？
- canary 和生产环境的服务端口是一样的吗？

136

- 生产环境的部署是一步到位还是循序渐进的？
- 对于紧急情况，是否存在直接跳过 staging 和 canary 阶段的情况？

服务依赖

- 微服务的依赖项都有哪些？
- 微服务的客户端都有哪些？
- 微服务如何缓解依赖失效所带来的影响？
- 对于每个依赖项，是否都有备份、替代服务、回退方案或防御性缓存？

路由和服务发现

- 微服务的健康检测可靠吗？
- 健康检测是否准确地反映微服务的健康状态？
- 健康检测是否运行在独立的通道上？
- 是否使用了回路断路器来防止不健康的微服务发出请求？
- 是否使用了回路断路器来防止生产环境的业务流量被发送到不健康的主机或服务上？

服务和端点的解除

- 是否有解除微服务的相关流程？
- 是否有解除微服务 API 端点的相关流程？

伸缩性和高性能

增长规模

- 微服务的质的增长规模是怎样的？
- 微服务的量的增长规模是怎样的？

资源的有效利用

- 微服务是运行在专门的硬件上还是共享的硬件上？
- 是否使用了资源隔离技术？

资源感知

- 微服务的资源需求是怎样的（CPU、内存，等）？
- 每个微服务实例能够处理多少流量？
- 每个微服务实例需要多少 CPU？
- 每个微服务实例需要多少内存？
- 微服务还需要其他的资源吗？
- 微服务的资源瓶颈在哪里？
- 微服务是否需要被横向或纵向扩展，或者两者兼顾？

容量规划

- 容量规划是否基于调度进行？
- 新硬件多久能够到位？
- 申请硬件的频率是怎么样的？
- 是否根据优先级为微服务分配硬件？
- 容量规划是自动化还是手工操作的？

依赖项的伸缩

- 微服务的依赖项有哪些？
- 这些依赖项是否具备了伸缩性和高性能？
- 依赖项能否随着微服务进行伸缩？
- 依赖项的所有者是否做好随微服务进行伸缩的准备？ 138

流量管理

- 是否很好地了解了微服务的流量模式？
- 是否根据流量模式来安排服务的变更？
- 流量模式的急剧变化（特别是流量爆发）是否被小心地处理了？
- 在服务失效以后，流量是否能够被恰当地重新路由到其他数据中心？

任务处理

- 微服务所使用的编程语言是否具备伸缩性和高性能？
- 微服务在处理请求时是否存在伸缩性和性能方面的限制？
- 微服务在处理任务时是否存在伸缩性和性能方面的限制？
- 微服务团队的开发人员是否了解他们的服务是如何处理任务的，处理任务的效率

是怎样的，以及当任务和请求数量增加时他们的服务将会如何应对？

可伸缩的数据存储

- 微服务是否以可伸缩和高性能的方式处理数据？
- 微服务需要存储什么类型的数据？
- 微服务的数据需要怎样的 schema ？
- 每秒需要处理多少个事务？
- 微服务需要更高的读写性能吗？
- 微服务是读密集、写密集还是两者兼顾？
- 微服务的数据库可以横向或纵向扩展吗？它是可复制或者可分区的吗？
- 微服务使用的是专门的还是共享的数据库？
- 微服务是如何存储和处理测试数据的？

容错和灾备

避免故障点

- 是否存在单点故障？
- 是否存在多点故障？
- 故障点是否能够被移除，或者需要对它们进行缓解吗？

故障场景

- 是否所有可能的故障场景都已被识别出来？
- 有哪些横跨整个生态系统的常见故障？
- 硬件层有哪些故障会影响到这个微服务？
- 通信层和应用平台层有哪些故障会影响到这个微服务？
- 哪些依赖项故障会影响到这个微服务？
- 哪些内部故障会拖垮这个微服务？

弹性测试

- 这个微服务是否通过了适当的 lint 测试、单元测试、集成测试和端到端的测试？
- 这个微服务是否经过合格的负载测试？
- 是否通过混沌测试对所有可能的故障场景进行了测试？

故障检测和修复

- 在组织里是否有标准化的事故处理流程？
- 这个微服务的故障是如何影响业务的？
- 是否对故障进行了清晰的分级？
- 是否有清晰的缓解策略？
- 当发生事故时，团队是否遵循事故处理的五个步骤？

监控

140

关键性的度量指标

- 这个微服务有哪些关键性的度量指标？
- 有哪些主机级别和基础设施级别的度量指标？
- 有哪些微服务级别的度量指标？
- 这些关键性度量指标都被监控起来了吗？

日志

- 这个微服务需要把哪些信息记录到日志里？
- 这个微服务是否记录了重要的请求消息？
- 日志是否准确地反映了微服务在各个时间点的状态？
- 这个日志方案是否具有伸缩性和高性价比？

仪表盘

- 这个微服务是否有仪表盘？
- 仪表盘是否简单易懂？是否所有的关键性度量指标都展示在了仪表盘上？
- 是否能够从仪表盘上看出这个微服务是否运行正常？

告警

- 是否每个度量指标都设有告警？
- 是否所有告警都设置了合适的阈值？
- 告警阈值设置是否恰当，以便在发生中断之前触发告警？
- 告警是否具有可操作性？
- 运行手册里是否包含了用于诊断、缓解和解决问题的排查步骤？

141

轮班待命

- 是否有一个专门的轮班待命机制用于微服务的监控?
- 每次待命排班是否有至少两个开发人员参与?
- 这个待命流程是否在整个工程组织内进行标准化?

文档化和理解

微服务文档

- 微服务的文档是否被集中存放在一个公开的、人们容易访问到的地方?
- 文档是否方便搜索?
- 微服务发生重要变更时，文档是否也得到了相应的更新？
- 文档是否包含了微服务的描述?
- 文档是否包含了架构图?
- 文档是否包含了待命信息?
- 文档是否包含了重要信息的链接?
- 文档是否包含了开发上手指南?
- 文档是否包含了微服务请求消息流、端点和依赖项的信息?
- 文档是否包含了运行手册?
- 文档是否包含了问答章节?

微服务理解

- 团队里的每个开发人员是否都能回答与他们的微服务的生产就绪相关的问题?
- 微服务是否遵循了一系列原则和标准?
- 对于新的微服务，是否有 RFC 流程?
- 已有的微服务是否经常得到评审和审计?

142

- 是否每个微服务团队都举行架构评审?
- 是否有生产就绪的审计流程?
- 是否有生产就绪路线图用于把微服务带向生产就绪的状态?
- 生产就绪标准是否推动了组织的 OKR？
- 生产就绪流程是自动化的吗?

术语表 143

可操作的告警

可操作的告警被触发时，轮班待命人员可以根据排查步骤对其进行诊断、缓解，并将其解决。

告警

在服务的关键性度量指标达到严重或警告的阈值时，用于通知待命的开发人员。

告警阈值

为关键性度量指标设置的一个数量值，可以是固定的，也可以是变化的，用于标识指标所处的状态：正常、警告或严重。指标达到阈值会触发告警。

应用平台层

微服务生态系统的第 3 层，包含内部自助工具、开发环境、测试工具、打包工具、构建工具、发布工具、部署管道、微服务级别的日志，以及微服务级别的监控。

应用程序接口（API）

为客户端定义的良好的接口，客户端可以通过编程的方式与服务发生交互，只要向固定的端点发送请求。

架构图

微服务架构的高层次的可视化表示。

架构评审

用于评估、理解和改进微服务架构的组织流程。

裸机

与从云服务提供商那里租来的硬件不同，裸机指的是组织自有的服务器，需要自己维护。

canary

部署管道的第二个阶段，使用一小部分服务器处理生产流量（生产流量的 2% 到 5%），在新版本经过 staging 测试之后，并在进入生产环境之前对其进行测试。

生产候选版本

经过开发周期的代码初步检查、单元测

试、集成测试和端到端测试，准备进入部署管道。

容量规划

资源分配调度和计划。

云服务提供商

提供硬件资源租用服务的厂商，如 Amazon Web Services（AWS）、Google Cloud Platform（GCP）和 Microsoft Azure，这些资源可以方便地通过安全网络访问。

144

代码测试

检查代码的语法和格式，测试微服务的组件个体，测试组件间的行为，测试微服务在依赖关系链里的行为；包括 lint 测试、单元测试、集成测试和端到端测试。

通信层

微服务生态系统的第 2 层；包括网络、DNS、RPC 框架、端点、消息服务、服务发现、服务注册和负载均衡。

并发

应用或微服务把每个任务拆分成更小的执行单元，而不是使用单个进程完成所有的工作；并发是伸缩性的基础。

持续集成

一个自动化流程，基于调度进行持续的集成、测试、打包、构建。

康威定律

由 Melvin Conway 提出的非正式软件架构“定律”，该定律指出，组织的交互模式决定了公司的产品架构；另有反康威定律。

仪表盘

一个可视化的图形界面，展示在内部的网站上，包含应用、微服务或系统的健康状况、状态、行为和关键性度量指标。

停运

微服务或 API 端点的退役过程，它们对于上游服务（客户端）来说不再可用。

专门硬件

只运行单个应用、微服务或系统的服务器，或者只保存它们的数据的数据库。

防御性缓存

将微服务下游依赖项的数据缓存起来，当下游依赖项不可用时，避免稳定性和可靠性受到影响。

依赖项

接收请求消息的微服务是发送请求消息的微服务的依赖项；微服务依赖的软件包也叫依赖项；微服务依赖的外部服务（第三方）也叫依赖项。

部署

将构建包发送到服务上并启动服务的过程。

部署管道

构建包需要经过的三个部署阶段（staging、canary 和生产环境）。

弃用

微服务或它的端点不再被维护，不建议上游服务（客户端）使用。

开发速度

开发团队迭代、发布新特性和部署的速度。

开发周期

应用、微服务或系统的整个开发流程。

开发环境

一个包含了工具、环境变量和流程的系统，开发者用它来开发微服务。

端点

145

在本书中，这个术语指的是微服务的固定 API 端点（HTTP、Thrift 等）。

外部故障

在微服务生态系统的下面三层所发生的故障。

full staging

在部署管道的 staging 阶段运行生产环境的完全镜像。

增长规模

用于衡量应用、微服务或系统是如何进行伸缩的；增长规模包括质的增长规模和量的增长规模。

硬件层

微服务生态系统的第 1 层；包括物理服务器、操作系统、资源隔离和资源抽象、配置管理、主机级别的监控和主机级别的日志。

硬件资源

见“资源”的定义。

水平伸缩

通过增加更多的服务器（或其他硬件资源）对应用或系统进行伸缩。

主机和基础设施度量指标

微服务生态系统底下三层（硬件层、通信层和应用层）的关键性度量指标。

主机平价

两个独立的环境、系统或部署阶段（如 staging 阶段与生产阶段）有相同数量的主机。

基础设施

在这本书里，基础设施是指应用平台层和通信层的组合或者微服务生态系统底下的三个层（硬件层、通信层和应用平台层）。

集成测试

测试微服务的组件（组件是通过单元测试进行测试的）是如何在一起工作的。

内部故障

微服务的内部故障。

反康威定律

对康威定律进行了反转，该定律指出，产品架构决定了公司的组织结构。

关键性度量指标

应用、微服务或系统的属性，用于描述应用、微服务或系统的健康状况、状态和行为。

lint 测试

检查代码的语法和格式错误，是代码测试的一部分。

负载均衡

一种设备或服务，为多个服务器或微服务分发流量。

日志

记录一个应用、微服务或系统的事件。

微服务

小型的软件应用，为大型的系统提供单一的功能，可替换、模块化、独立开发、独立部署。

146

微服务生态系统

包含了微服务和基础设施的系统，可以分为 4 层，分别是微服务层、应用平台层、通信层和硬件层。

微服务层

微服务生态系统的第 4 层；包括微服务和微服务所有的相关配置。

微服务度量指标

微服务关键性度量指标，对于微服务生态系统里的每个微服务来说都是独一无二的。

监控

在较长的一段时间内观察和跟踪应用的状态、健康情况、行为或微服务的关键性度量指标。

单体应用

大型复杂的软件系统，它包含了所有相关代码和功能，被作为一个单独的应用进行维护、运行和部署。

轮班待命

一组开发人员或运维工程师，他们负责对应用、微服务或系统的告警、事故和故障进行响应、缓解和解决。

运行手册

微服务文档里的一个章节，包含事故和中断的响应流程，诊断、缓解和解决告警的排查步骤，以及用于调试和诊断微服务的帮助信息；待命的开发人员或运维工程师会使用运行手册。

运维工程师

运维工程师的主要职责是执行与运行软件应用相关的任务，包括系统管理员、技术运维、DevOps 和网站可靠性工程师。

中断

应用、微服务或系统的不可用期间（宕机）。

partial staging

部署管道的 staging 阶段使用的不是生产环境的完全镜像，但 staging 环境的微服务与生产环境的客户端、依赖项和数据

库发生交互。

分区

将每个任务拆分成更小的单元，这样就可以对它们进行并行处理；是伸缩性的基础。

生产

部署管道的最后阶段，这里有来自真实世界的流量；也指真实世界的流量和环境。

生产就绪审计

根据生产就绪检查列表对微服务进行生产就绪评估的过程。

生产就绪自动化

通过自动化和编程的方式检查微服务是否达到与生产就绪相关的要求，以此来确保微服务达到了生产就绪标准。

生产就绪检查列表

一个生产就绪标准列表，包含用于实现每个生产就绪标准的相关要求。

生产就绪路线图

一份文档，作为生产就绪流程的一部分，描述了把微服务带向生产就绪状态的步骤。

生产就绪评分

微服务的评分，根据微服务满足每项生产就绪标准程度计算得出的。

发布和订阅消息模型

一种异步消息模型，客户端订阅主题，当发布者向主题发布消息时，客户端就会收到消息。

质的增长规模

用于衡量应用、微服务或系统的增长与业务指标之间的关系；是增长规模的一种。

量的增长规模

对应用、微服务或系统增长的量化；将量的增长规模转化成一个可衡量的数量；是增长规模的一种；一般使用应用、微服务或系统能够处理的每秒请求数、每秒查询数或每秒事务数来表示。

远程过程调用（RPC）

通过网络对远程服务器的调用被设计成看起来像在调用本地过程一样；在微服务架构和大型分布式系统中被广泛应用。

仓库

一个集中式的归档中心，用于保存应用或服务的所有源代码。

请求消息流

请求消息从一个微服务发送到另一个微服务所经历的步骤。

请求和响应消息模型

147

一种消息模型，客户端发送一个请求到微服务（或消息代理服务器），服务返回相应的信息。

资源分配

在微服务生态系统里分配硬件资源。

资源瓶颈

应用、微服务或系统在使用资源时出现的伸缩性极限。

资源需求

应用、微服务或系统所需要的资源。

资源

对硬件（服务器）性能属性的抽象描述，比如 CPU、内存、网络等。

自助内部工具

微服务生态系统应用平台层的标准化工具，用于帮助开发人员在微服务生态系统上开发、部署和运行微服务。

服务发现

一个用于发现微服务实例的系统，确保流量被路由到正确的服务器上。

服务注册

一个数据库，记录了微服务生态系统里所有微服务和系统的端口和 IP 地址。

共享硬件

用于部署多个应用、微服务或系统的服务器，或存储数据的数据库。

单点故障（SPOF）

应用、微服务或系统的一部分，如果它发生故障，会拖垮整个应用、微服务或系统。

网站可靠性工程师（SRE）

148

大型公司的运维工程师，在工程组织内负责应用、微服务或系统的可靠性。

拆分单体

将大型单体应用拆分成一系列微服务。

staging

部署管道的第一个阶段，用于测试新的构建包，不处理生产流量；一般是生产环境的副本；包括 full staging 和 partial staging。

三层架构

一个基本的软件架构，由前端（客户端）、后端和数据存储组成。

单元测试

独立的小型测试，用于测试微服务代码单元；是代码测试的一部分。

垂直伸缩

通过增加资源（CPU、内存）对应用或系统进行伸缩。

索引†

†: 索引所列页码为本书英文版页码，请参照正文侧边用“⟨”表示的原书页码。

E

F

G

H

I

T

U

V

关于作者

Susan Fowler 是 Uber 的网站可靠性工程师，她致力于在 Uber 的所有微服务上推行生产就绪标准化，并参与到关键性业务团队中，帮助他们将微服务带向生产就绪状态。在加入 Uber 之前，她在一些初创公司的应用平台和基础设施平台上工作。她之前在宾夕法尼亚大学学习粒子物理学，研究超对称性，并为 ATLAS 和 CMS 检测器设计硬件。

关于本书封面

这本书封面上的动物是切叶蜂（leafcutter bee），属于切叶蜂科（Megachile）。切叶蜂科有 1500 多个品种，分布在世界各地。其中来自印度尼西亚的冥王切叶蜂被认为是世界上最大的切叶蜂，它们的体长可以达到 0.9 到 1.5 英寸。

切叶蜂的得名源于母蜂经常进行的一种活动，它们从叶子边缘切出整齐的半圆，并把这些切片带回巢中。它们的巢可以筑在凹陷处、地表的洞穴里，或者腐烂的木头里。它们的巢在 4 到 8 英寸之间，呈圆柱体状，叶子的切片被依次层叠排列。这种昆虫并非以群居的方式生活，尽管它们的巢挨得很近。

母蜂将巢分为很多独立的巢室，并在每个巢室里放置一个蜂卵和一个咀嚼过的花粉蜜球。这个花粉蜜球是幼虫的食物，根据推测，放置叶子切片的目的是为了防止花粉蜜球变干。

成年切叶蜂也以花粉蜜为食，它们在花朵中像游泳一样尽情地传粉（辛勤地传粉并将它们腹部的毛发裹满花粉），是辛勤的传粉者。母蜂通常需要 10 到 15 个来回才能安置好一个巢室，这个举动进一步提升了传粉效率。所以，切叶蜂很受花园和农场的欢迎，人们会用筑巢用的盒子和管子来吸引它们。

O'Reilly 封面上的动物大部分都濒临灭绝，它们都是很重要的动物。如果想了解如何保护动物，可以访问 *animals.oreilly.com*。

封面图片来自 Lydekker 的 *Royal Natural History* 一书。

O'REILLY®

生产微服务

对于已经采用了微服务架构的组织来说，最重要的问题是缺乏架构标准、运维标准和组织标准。在将一个单体应用拆分成微服务或从头创建一个微服务生态系统之后，很多工程师不知道下一步该怎么做。在这本书里，Susan Fowler基于她在Uber对上千个微服务进行标准化的经验，为我们深入总结了一组微服务标准。你将学到如何设计稳定、可靠、可伸缩、高容错能力、高性能、可监控、文档化和具有灾备能力的微服务。

生产就绪标准包括如下项目。

- **稳定性和可靠性：开发、部署、添加和移除微服务；避免依赖项失效。**
- **伸缩性和高性能：学习如何提升微服务的效率。**
- **容错和灾备：通过实时、主动地制造失效来提升微服务的可用性。**
- **监控：学习如何监控、记录日志和展示关键性度量指标；建立告警和待命流程。**
- **文档化和理解：缓解微服务架构带来的问题，包括组织蔓延和技术债务。**

Susan Fowler是Uber的网站可靠性工程师，她的工作内容主要有两方面。一方面为Uber的微服务制订生产就绪标准，一方面帮助关键性业务团队把他们的微服务带向生产就绪状态。在加入Uber之前，她在宾夕法尼亚大学学习粒子物理学，随后在一些初创公司工作，负责应用平台和基础设施。

"我相信这本书注定会成为微服务设计和运维事实上的参考标准，仅是生产就绪标准检查清单就值得购买本书！"

——Daniel Bryant
OpenCredo首席科学家

"实现微服务架构是非常困难的，特别是从运维角度来看。这本书将带你了解是什么造就了生产就绪的微服务。不管你在企业中的角色是什么，Susan都会让你洞见如何构建一个高效的微服务生态系统。"

——Mark Richards
独立咨询顾问

SYSTEMS ARCHITECTURE / MICROSERVICES

图书分类：系统架构/微服务

策划编辑：张春雨
责任编辑：刘舫

Broadview®
www.broadview.com.cn

O'Reilly Media, Inc.授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao SAR and Taiwan)

ISBN 978-7-121-32433-8

定价：55.00元